6

LA GESTIONE DEI FILES

Se fin qui ci siamo occupati prevalentemente di ciò che succede all'interno della memoria, è ora il momento di vedere più da vicino come scrivere un programma che comunichi con l'esterno, sia interagendo tramite terminale – cosa che in piccola misura abbiamo già fatto – sia scrivendo o leggendo files di dati.

Comincerò quindi col descrivere come si trattano dati scritti in formato leggibile (o formato ASCII), per poi passare alla scrittura di dati nella forma binaria che hanno in memoria. Il tutto senza trascurare di aprire un po' lo sportellino del disco e darci un'occhiata rapida.

Ma non pensare che quello che trovi qui sia tutto sui files: i sistemi mettono a disposizione un gran numero di strumenti per gestirli. Io intendo limitarmi ai metodi più immediati e comuni. Quando avrai acquisito familiarità sufficiente o avrai problemi più sofisticati potrai tu stesso cercare altre informazioni nei manuali: troverai modo di fare tutti gli errori che vuoi, e anche di imparare.

6.1 La comunicazione con tastiera e schermo

La scrittura su schermo e la lettura da tastiera sono tipici esempi di trattamento di files *formattati.* [1] Il termine *formattato* sta ad indicare un file il cui contenuto è leggibile da un normale rappresentante dell'umanità — da intendersi: una persona che abbia imparato a leggere, ma non è ancora traviata al punto da leggere i files binari del calcolatore. Si parla spesso anche di *file in formato ASCII*, intendendo un file il cui contenuto è costituito da soli caratteri stampabili del codice ASCII, ed eventualmente organizzato in righe. Il punto è che la memoria conserva le informazioni in formato binario — quello di cui abbiamo discusso nel capitolo 4. I dati inseriti da tastiera o quelli che compaiono sullo schermo – fatti salvi i dati grafici visualizzati

[1] Come avviene in altri casi, anche il termine formattato è una traduzione folcloristica, ma ormai profondamente radicata nell'uso, dell'inglese *formatted.*

Tavola 6.1. I formati di conversione più comuni per la lettura o scrittura di dati interi o caratteri ASCII

Formato	Tipo	Note
`%hhd`	`char`	in decimale, con eventuale segno
`%hhu`	`unsigned char`	in decimale, senza segno
`%hho`	`char`	in ottale
`%hhx`	`char`	in esadecimale
`%hd`	`short int`	in decimale, con eventuale segno
`%hu`	`unsigned short int`	in decimale, senza segno
`%ho`	`short int`	in ottale
`%hx`	`short int`	in esadecimale
`%d`	`int`	in decimale, con eventuale segno
`%u`	`unsigned int`	in decimale, senza segno
`%o`	`int`	in ottale
`%x`	`int`	in esadecimale
`%ld`	`long int`	in decimale, con eventuale segno
`%lu`	`unsigned long int`	in decimale, senza segno
`%lo`	`long int`	in ottale
`%lx`	`long int`	in esadecimale
`%Ld`	`long long int`	in decimale, con eventuale segno
`%Lu`	`unsigned long long int`	in decimale, senza segno
`%Lo`	`long long int`	in ottale
`%Lx`	`long long int`	in esadecimale
`%c`	`char`	singolo, in formato ASCII
`%s`	`char[]`	una stringa di caratteri

come immagini – devono essere invece leggibili: c'è di mezzo una conversione che viene detta *formattazione* — da non confondersi con la formattazione dei dischi.

Le funzioni che operano su tastiera e schermo sono fondamentalmente `scanf` e `printf` che già hai avuto modo di usare. Qui le vediamo in modo un po' più approfondito.

6.1.1 La lettura da tastiera

L'istruzione l'hai già vista:

`scanf(`⟨*formato*⟩`,`⟨*argomenti*⟩`);`

dove il formato è una stringa di caratteri che specifica come debba essere eseguita la conversione dei dati, e gli argomenti sono gli indirizzi delle celle di memoria dove vanno depositati i dati.

Ti richiamo in ordine alcune cose già viste, in particolare nel paragrafo 2.3.1.

- Ogni specifica di conversione è formata dal carattere `%` seguito da ulteriori specifiche; qui ti indico le più comuni.

Tavola 6.2. I formati di conversione più comuni per la lettura di dati reali.

Formato	Tipo	Note
`%e`	`float`	in decimale, con eventuale segno
`%le`	`double`	in decimale, con eventuale segno
`%Le`	`long double`	in decimale, con eventuale segno

- Ad ogni specifica di conversione deve corrispondere l'indirizzo di un dato.

Se hai messo più indirizzi che specifiche di conversione, la funzione leggerà solo un numero di dati pari a quello delle specifiche, lasciando gli altri nello stato in cui si trovavano. Se hai messo più stringhe di conversione che indirizzi di dati andrà ancor peggio: la funzione continuerà a cercare indirizzi nella tabella degli argomenti proseguendo in ordine, senza accorgersi che la tabella è finita. L'invasione di memoria è praticamente garantita.

Veniamo ai formati di conversione più comuni. Nella tavola 6.1 trovi un elenco abbastanza dettagliato per le conversioni di dati interi, di caratteri singoli o di stringhe di caratteri. La puoi riassumere così:

- I dati di tipo intero si convertono coi formati `%d` (con segno), `%u` (senza segno), `%o` (ottale) e `%x` (esadecimale); alle quattro lettere qui sopra si devono premettere i caratteri `hh` per dati `char`, `h` per `short int`, `l` per `long int` e `L` per `long long int`.
- I caratteri ASCII singoli si leggono col formato `%c`;[2]
- Le stringhe di caratteri si leggono col formato `%s`; la stringa termina col tasto <Return>, e devi passare l'indirizzo di un vettore di lunghezza sufficiente a contenere tutti i caratteri battuti più il `NUL` che chiude la stringa. È possibile – e fortemente consigliato – usare il formato `%`n`s`, e passando l'indirizzo di un vettore `char` di lunghezza almeno $n+1$. Ad esempio, `%23s` significa che voglio leggere al più 23 caratteri, escluso il `NUL` finale, e che mi impegno ad affermare che il vettore destinatario può contenere almeno 24 caratteri.

I formati di conversione per dati di tipo reale sono riassunti nella tavola 6.2. Il formato `%e` deve essere usato per dati `float`; la lettera `e` deve essere preceduta da `l` per dati `double` e `L` per `long double`. La lettera `e` può essere sostituita da `f` o `g`.

Le tabelle non sono esaustive: la pagina di manuale che descrive la funzione `scanf` è quanto mai ricca di possibilità. Se un giorno ti capitasse di domandarti: "ma non posso costringere `scanf` ad accettare solo caratteri alfabetici?" sappi che la risposta è sí. Il manuale ti spiega come fare.

[2] Presta ben attenzione: il dato è il carattere ASCII stesso, senza conversione. Ad esempio, se leggi `1` in formato `%d` in memoria ci finisce il valore 1 in binario; se lo leggi in formato `%c` in memoria ci finisce il codice ASCII del carattere `1`, che vale 61_8.

E, come sempre, parliamo anche di trabocchetti; ce ne sono, e qui sono più subdoli che altrove.

- L'uso dei diversi formati con le specifiche `hh`, `h`, `l` o `L` è praticamente obbligatorio: se il dato è `short int` devi usare `%hd` (o `%hu`, o `%ho` o `%hx`) e devi ben guardarti dall'usare `%d` o simili. Il motivo è ben semplice: `scanf` riceve solo un indirizzo, e la sola informazione che ha sulla lunghezza del dato è quella letterina `h`, `l` o `L` che tu metti nel formato. Se sbagli, `scanf` riempie una cella della lunghezza che gli hai indicato tu nel formato. Riesci ad immaginare le conseguenze? Prova!
- La stessa osservazione vale, *mutatis mutandis*, per i dati reali.
- L'inserimento di caratteri o scritte più o meno strane che non siano i formati ammessi – e sempre preceduti da `%` – può essere causa di gravi emicranie per chi non si rende conto degli effetti collaterali. Ad esempio, se io scrivo
  ```
  scanf("Dimmi un numero: %d",&j);
  ```
  la funzione `scanf` andrà a cercare la stringa `Dimmi un numero:` prima di prendere in considerazione il formato di conversione `%d`. Detto in altri termini, si rifiuterà di iniziare la lettura con qualcosa che non sia proprio quella scritta! È un errore che ho visto commettere più volte: immagino che il programmatore di turno intendesse – incautamente – chiedere a `scanf` di scrivere `Dammi un numero:` sul terminale. Non è così che si ottiene lo scopo: per scrivere si usa `printf`, non `scanf`.

6.1.2 La scrittura sullo schermo

La funzione di uso più comune è

`printf(`⟨*formato*⟩`,`⟨*argomenti*⟩`);`

dove il formato specifica cosa tu voglia veder scritto sullo schermo, incluse le conversioni di dati dal formato binario a quello leggibile.

Il formato è una stringa di caratteri che la funzione riporta sullo schermo tali e quali fin che incontra un `%`. Quest'ultimo carattere significa che vuoi la conversione di un dato, e deve essere seguito da ulteriori specifiche, simili a quello della funzione `scanf`. Nella forma più elementare i formati di conversione per interi, caratteri singoli o stringhe di caratteri sono gli stessi della `scanf`, e li trovi elencati nella tavola 6.1. Ci sono però delle varianti.

Il formato generale per un formato di conversione di interi è

`%`⟨*campo*⟩⟨*conversione*⟩ ,

dove ⟨*conversione*⟩ è uno dei formati indicati nella tavola 6.1, mentre ⟨*campo*⟩ specifica quanto spazio ha a disposizione per scrivere il numero. Ad esempio, `%6d` significa che voglio che usi lo spazio di 6 caratteri per scrivere il numero. Se occorrono meno di 6 cifre, aggiunge degli spazi in testa.

La forma generale è tipicamente utile per stampare tabelle incolonnate. Ad esempio, supponiamo di avere un vettore `vet` di 100 numeri interi non negativi, di cui sappiamo che occupano al massimo 4 cifre ciascuno, e di volerli stampare in una tabella di 10 colonne di ampiezza 5 caratteri ciascuna.

Al primo momento forse resterai perplesso, vista la scarsità di strumenti che hai a disposizione. Ma se rifletti qualche istante ti renderai conto che c'è una soluzione semplice. Prima di proseguire prova a pensarci; poi confronta la tua soluzione con quella che trovi qui sotto.

Eccoti in breve l'algoritmo:

(i) costruisci un ciclo con un contatore `j` che va da `0` a 99;
  (i.a) se `j` è multiplo di 10 scrivi un ritorno a capo;
  (i.b) stampa il `j`–esimo elemento del vettore senza aggiungere il ritorno a capo;

(ii) scrivi un ultimo ritorno a capo, se serve.

Ed ora eccoti il frammento di programma:

```
for(j=0; j<100; j++) {
  if((j%10) == 0) printf("\n");
  printf("%5d");
}
printf("\n");
```

Noterai che prima della tabella comparirà una linea vuota. Se ti sembra di troppo, sostituisci la seconda istruzione con

```
if((j%10)==0 && j!=0) printf("\n");
```

e l'avrai eliminata. Se provi ad eseguire un programma che contenga questo frammento di codice vedrai comparire sul terminale qualcosa come

```
    2    3    5    7   11   13   17   19   23   29
   31   37   41   43   47   53   59   61   67   71
   73   79   83   89   97  101  103  107  109  113
  ...  ...  ...  ...  ...  ...  ...  ...  ...  ...
  463  467  479  487  491  499  503  509  521  523
```

Osserva che tutti numeri sono incolonnati a destra, e che la funzione `printf` ha sostituito le cifre mancanti con degli spazi.

In questa stampa i primi 10 elementi del vettore stanno sulla prima riga, i 10 successivi sulla seconda, &c. Se volessi stampare i numeri in colonna dovresti modificare le istruzioni inserendo un doppio ciclo. Te lo lascio come esercizio.

Veniamo alla conversione di numeri reali. Rispetto agli interi vi sono tre differenze: la corrispondenza tra formato e tipo di dato, la forma generale che prevede anche il numero di cifre di precisione ed i tre diversi formati di conversione `f`, `e` e `g`.

Per i formati devi guardare la tavola 6.3. Noterai una stranezza: le specifiche per dati `float` e `double` sono identiche. Non è un refuso. Il motivo sta in una convenzione del linguaggio C: tutti i dati `float` specificati come argomento di una funzione e passati per valore vengono convertiti in `double` prima di essere copiati nella tabella degli argomenti.[3]

---

[3] È una convenzione un po' strana ed arbitraria, ma esiste fin dagli albori nel

Tavola 6.3. I formati di conversione più comuni per la scrittura di dati reali.

Formato	Tipo	Note
`%f`	`float`	in notazione senza esponente
`%f`	`double`	in notazione senza esponente
`%Lf`	`long double`	in notazione senza esponente
`%e`	`float`	in notazione esponenziale
`%e`	`double`	in notazione esponenziale
`%Le`	`long double`	in notazione esponenziale
`%g`	`float`	come `f` o `e`, dipende dalla lunghezza
`%g`	`double`	come `f` o `e`, dipende dalla lunghezza
`%Lg`	`long double`	come `f` o `e`, dipende dalla lunghezza

La forma generale è
`%`⟨*campo*⟩`.`⟨*precisione*⟩⟨*conversione*⟩ ,
dove ⟨*campo*⟩ è ancora l'ampiezza del campo disponibile per la conversione, ⟨*precisione*⟩ è il numero di cifre di precisione, ossia il numero di cifre che compare dopo il punto decimale.

La scelta del formato `f` significa che voglio la scrittura abituale ⟨*parte intera*⟩`.`⟨*parte decimale*⟩ con il numero di cifre decimali specificato dalla precisione. Va da sè che l'ampiezza del campo assegnato deve tenere conto del numero di cifre decimali richieste come ⟨*precisione*⟩, del punto di separazione della parte decimale, dell'eventuale segno e del numero di cifre della parte intera. Conclusione immediata: il formato `f` è utile quando i numeri da stampare sono in un intervallo abbastanza ristretto.

Il formato `e` indica che voglio la notazione scientifica: una cifra per la parte intera, tante cifre decimali quante ne ho richieste come ⟨*precisione*⟩, la lettera `e`, l'esponente di 10 con segno; per intenderci, se specifico `%12.6e` il numero e – la base dei logaritmi naturali – viene scritto come `2.718282e+00`. Nella specifica dell'ampiezza del campo si deve tener conto delle cifre decimali richieste, del punto, della cifra della parte intera, della lettera `e`, del segno dell'esponente e di due cifre di esponente; dunque l'ampiezza minima deve essere almeno ⟨*precisione*⟩ $+\,7$ per numeri che possono avere segno negativo; almeno uno spazio in più è utile come separazione dalle scritte precedenti. In altre parole, un formato del tipo `%10.7e` è assurdo. Il formato `e` è

linguaggio C, e non è mai stata modificata. Del resto una convenzione simile esiste anche per gli interi: un dato intero di lunghezza inferiore ad `int` viene convertito in `int` prima di essere inserito nella tabella degli argomenti. Attenzione però: queste convenzioni non si applicano ai dati passati per indirizzo. In quest'ultimo caso nella tabella degli argomenti compare l'indirizzo del dato; il dato resta al suo posto in memoria, e conserva la sua lunghezza. Per questo motivo occorre fare molta attenzione alla lettura con `scanf`, mentre l'uso di `printf` è, tutto sommato, meno pericoloso.

comodo quando si debbano stampare numeri di ordini di grandezza molto variabili — il che del resto giustifica l'uso della notazione esponenziale in ogni campo della scienza.

Il formato `g` è un'indicazione molto stretta: si scrive sempre un numero di cifre pari a ⟨*precisione*⟩ nel campo di ampiezza ⟨*campo*⟩. Se lo spazio è sufficiente si ricorre al formato `f`, altrimenti si passa al formato `e`.

Non mi dilungo oltre nel descrivere tutti i casi possibili: non sono infiniti solo perché il numero di stati possibili del calcolatore, per quanto grande, resta pur sempre finito. Ma sono comunque troppi. Qualunque domanda tu abbia, la risposta è una sola: prova! E se per caso ti interessasse fare qualcosa che non è descritto in questo breve paragrafo leggi la pagina del manuale: è una miniera di informazioni, ed un eccellente esercizio di computerese.

6.2 I files

Il trasferimento di dati tra memoria e periferici non si riduce certo alla lettura di dati da tastiera o alla scrittura sullo schermo. C'è ben altro, ovviamente: la stampante, il modem, lo scanner, il mouse, il dischetto, il lettore di CD, le casse acustiche, i dischetti floppy, il disco rigido, tanto per limitarsi alle periferiche più comuni. Ha l'aria di essere un gran guazzabuglio, ma . . .

6.2.1 Che diavolo è un file?

È ormai pratica comune nella scrittura dei sistemi operativi – e Linux certo non fa eccezione – limitarsi a considerare tutto come un *file*. Il termine *file*, da tradursi letteralmente con schedario o archivio, identifica un pacchetto di informazioni di qualunque natura che possa essere trasferito da un componente del computer all'altro. Il contenuto poco importa: in fondo, si tratta pur sempre di bit.

Ciò che differenzia un file dall'altro è, in prima battuta, il periferico a cui è destinato. La tastiera può trasmettere solo dei codici numerici che qualcuno –il driver, o qualche altro programma – interpreta come caratteri; lo schermo può ricevere sequenze di caratteri da stampare, o sequenze di dati grafici da visualizzare come figura, [4] la stampante può ricevere sequenze di caratteri ASCII, o sequenze più complesse che vengono interpretate come testo o grafico; la scheda audio riceve una sequenza di dati digitalizzati che vengono tradotti in un segnale analogico, spedito a sua volta alle casse acustiche. Potrei continuare, ma mi sembra superfluo.

Più interessante è chiedersi: ma cosa differenzia i dati destinati alla scheda grafica da quelli destinati alla stampante? La risposta è sorprendentemente semplice: il modo di interpretare i dati da parte del destinatario.

---

[4] Per essere precisi, l'oggetto che viene controllato direttamente è la scheda grafica: il contenuto della memoria della scheda viene trasferito tale e quale sullo schermo.

Se il destinatario finale è l'uomo è inutile inviargli direttamente i dati digitalizzati di una canzone di Lucio Dalla: non li capirebbe; è indispensabile l'intervento di una scheda audio e delle casse acustiche.

La prima differenza rilevante è il modo in cui i dati vengono trasferiti ed utilizzati. Alcuni periferici – la maggior parte – sono in grado solo di ricevere o inviare una successione di bytes, tutti in fila, in un ordine ben definito: si dice che il flusso di dati è di tipo *sequenziale*, o si parla tout court di *file sequenziale*. Altri sono in grado di gestire il pacchetto di dati che costituisce un file in modo più libero: inviarne una parte, ripetere, saltare un blocco, ritornare all'inizio, &c. Si parla di *files ad accesso casuale*, o *random*. I rappresentanti principali di quest'ultima categoria sono i dischi.

La seconda differenza riguarda il contenuto. Alcuni files contengono solo caratteri stampabili: vengono detti files *ASCII*, e tutto sommato abbiamo già detto molto su questi. Esempio tipico: la tastiera ed il terminale. Altri files contengono dati destinati ad un particolare periferico: esempi tipici sono i files stampabili di tipo *PDF (Portable Document Format)* o *PostScript* – due tipi ormai molto diffusi perché permettono una gestione abbastanza ricca del contenuto: diverse fonti di scrittura, impaginazione, tabelle, grafici &c. Altri ancora contengono testi impaginati in modo più o meno sofisticato, ma modificabili da un programma che sia in grado di gestirli — Staroffice o Word, per fare solo un paio di esempi. Mi sembra inutile continuare: non farei altro che citare una lunga serie di sigle più o meno incomprensibili ai non addetti.

Solo un consiglio: quando senti parlare di un formato compresso *zip* o *gzip*, o di un formato musicale *mp3*, o di un formato grafico *jpg* o *gif* o *tiff*, o di qualunque altra amenità del genere non lasciarti intimidire. Le informazioni che ti interessano sono tipicamente tre: che tipo di dati ci sono in quel file (documenti, musica, grafica &c), quale programma è in grado di leggerli (o, più raramente, scriverli), e a quale tipo di periferico sono destinati — è inutile inviare un file musicale ad una stampante, o un file PostScript ad una scheda audio. E, soprattutto, ricorda: se tu avessi la pazienza di cercare le informazioni tecniche sulla struttura interna del file e di metterti a programmare saresti perfettamente in grado di creare tu stesso files col formato che vuoi. Spesso i venditori ti fanno pagare in moneta sonante la tua pigrizia — o la tua ignoranza.

6.2.2 *Come accedere ad un file*

Le regole del gioco sono, tutto sommato, semplici. Se vuoi scrivere un programma che legge o scrive un file devi eseguire, nell'ordine che ti indico qui sotto, alcune operazioni.

(i) Devi informare il sistema che vuoi accedere al file. L'operazione viene chiamata *apertura*, o *open*, del file ed è svolta dalla funzione `fopen`.

(ii) Fin che il file è aperto puoi chiamare delle funzioni che eseguono il trasferimento di dati da o verso il file.

(iii) Quando hai finito devi informare il sistema che non intendi più accedere al file. L'operazione viene detta *chiusura*, o *close*, del file; e viene svolta dalla funzione `fclose`. [5]

Il tipo di dati da trasferire, come ti ho spiegato, dipende dal periferico che stai usando. A seconda del tipo di dati potrai anche scegliere, in generale, la funzione più comoda tra le molte offerte tipicamente dalle librerie di sistema. Nel resto di questo paragrafo non mi riprometto di farti una panoramica completa delle funzioni: sarebbe troppo lungo, e tutto sommato inutile. Mi dilungherò un po' invece sui files di uso più frequente: quelli che si trovano su disco. Ma prima di discutere in dettaglio quali funzioni ti possano essere utili lascia che ti dia ancora alcune informazioni.

6.2.3 Files formattati e binari

Dovresti già sapere che i files su disco sono contraddistinti da un *nome*, e che i nomi sono contenuti in *directories* (o elenchi, o cartelle). Dovresti anche aver imparato a svolgere alcune operazioni fondamentali per la gestione: creare un directory, elencarne il contenuto, spostarti da un directory all'altro, copiare un file, cambiarne il nome o la posizione entro la struttura dei directories, rimuovere un file o un directory, &c. Se non hai ancora familiarità sufficiente con queste operazioni rischi di perdere molte delle informazioni che sto per passarti.

La distinzione fondamentale è tra files *binary* e files *formattati*. Binario è un file che contiene una copia esatta delle informazioni che sono state create in memoria, o che vi devono essere trasferite; il trasferimento avviene semplicemente byte–a–byte, senza modifiche. Formattato è un file che contiene dati che necessitano di un procedimento di conversione prima di passare all trasferimento.

Hai già visto almeno tre esempi di files formattati: la tastiera, lo schermo del terminale ed i files nei quali hai scritto i sorgenti dei tuoi programmi C. Come ti ho spiegato nel primo paragrafo devi usare le funzioni `scanf` (scan formatted) per la lettura da tastiera, e `printf` (print formatted) per la scrittura sul terminale. Ma, ... qual è il nome? Il sistema Linux (ed in generale i sistemi *Unix*) assegnano automaticamente i nomi `stdin` al file di lettura standard, `stdout` al file di scrittura standard e `stderr` al file destinato come standard a ricevere le segnalazioni di errore. A meno di altre indicazioni, `stdin` viene assegnato alla tastiera, e `stdout` e `stderr` allo schermo del terminale: per questo hai potuto impunemente usare le funzioni `scanf` e `printf` senza preoccuparti di aprire i files coinvolti. Per quanto riguarda i sorgenti

[5] L'abitudine di tralasciare la chiusura dei files che sono stati aperti è abbastanza diffusa tra i programmatori: il sistema comunque provvede a chiuderli tutti alla fine dell'esecuzione del programma. È tuttavia buona regola provvedere direttamente alla chiusura quando il files non servono più: si risparmiano risorse di sistema, e si diminuisce il rischio di errore.

dei programmi, tu non hai ancora imparato come scriverli o leggerli da un tuo programma, ma hai fatto abbondante uso del *text editor* per prepararli o correggerli.

Di files non formattati, contrariamente a quanto forse pensi, ne conosci già diversi: i directories, i files contenenti moduli rilocabili, le librerie di sistema, i files eseguibili. Questi non li scrivi tu, evidentemente: ci pensano il compilatore o il linker, o altri programmi forniti col pacchetto di sviluppo.

6.2.4 Come tratto i files formattati?

Tieni ben presente lo schema che ti ho spiegato nel paragrafo 6.2.2: devi aprire il file, leggerci o scriverci, e poi chiuderlo. Eccoti i frammenti di istruzioni che ti servono.

Per l'apertura:

```
FILE *fp;
fp = fopen(⟨nome⟩,⟨accesso⟩);
```

Qualche spiegazione.

- La dichiarazione `FILE *fp` significa che `fp` è una variabile destinata a contenere l'indirizzo di una tabella di dati che il sistema userà per accedere al file. Non è poi tanto diversa da `int` o `double`: il compilatore associa al nome `fp` una cella di memoria grande quanto basta per contenere un indirizzo. Il nome `fp` è di fantasia: puoi metterci quello che ti pare.
- L'istruzione `fp = fopen(...)` significa: voglio che mi sia reso disponibile il file ⟨*nome*⟩, e ci voglio accedere in modo ⟨*accesso*⟩. Il sistema associa a quel file una tabella di dati che gli serviranno per gestirne il contenuto, e restituisce l'indirizzo della tabella. Il compilatore predispone le istruzioni per memorizzare l'indirizzo in `fp`.
- Il campo ⟨*nome*⟩ può essere un qualunque nome di file: `"ciccio.mio"`, ben chiuso tra virgolette perché si tratta di una stringa di caratteri, va benissimo. Un vettore di tipo `char` che contenga la stringa del nome chiusa da un `NUL` va altrettanto bene.
- Il campo ⟨*accesso*⟩ può assumere uno dei valori seguenti (le virgolette fanno parte della sintassi):
  - `"r"` (read): voglio accedere in lettura ad un file che esiste;
  - `"r+"` (read and write): voglio accedere in lettura ed in scrittura ad un file che esiste;
  - `"w"` (write): voglio un file nuovo su cui scrivere; se per caso il file esistesse già azzera il suo contenuto e riscrivilo;
  - `"w+"` (write and read): voglio un file nuovo su cui scrivere e leggere; se per caso il file esistesse già azzera il suo contenuto e riscrivilo;
  - `"a"` (append): voglio scrivere su un file che potrebbe anche esistere, aggiungendo dati in fondo al contenuto attuale; se per caso non esistesse creane uno nuovo;

`"a+"` (append and read): voglio scrivere su un file che potrebbe anche esistere, aggiungendo dati in fondo al contenuto attuale, e poi lo voglio anche leggere; se per caso non esistesse creane uno nuovo;

Veniamo alla lettura o scrittura. Un file formattato viene tipicamente letto o scritto in modo sequenziale, benché non sia proibito, ad esempio, riposizionarsi all'inizio. Per il riposizionamento vedi le istruzioni nel prossimo paragrafo: funzionano anche qui. L'accesso sequenziale funziona esattamente come sulla tastiera o sul terminale; cambiano solo di poco e funzioni:

◦ `fscanf(fp,`⟨*formato*⟩`,`⟨*argomenti*⟩`);`

legge dei dati dal file, esattamente come se li avessi battuti sulla tastiera; il nome sta per *file scan formatted*;

◦ `fprintf(fp,`⟨*formato*⟩`,`⟨*argomenti*⟩`);`

scrive dei dati su file esattamente come li manderebbe sul terminale; il nome sta per *file print formatted*. Per il formato di lettura e scrittura valgono le stesse regole che ti ho enunciato nel paragrafo 6.1. Anzi, ti dirò di più: le funzioni `scanf(...)` e `printf(...)` sono del tutto equivalenti a `fscanf(stdin,...)` e `fprintf(stdout,...)`.

La chiusura del file:

`fclose(fp);`

dichiara al sistema hai finito di lavorare sul file; il sistema lo chiude, nel senso che completa le operazioni di scrittura eventuali in corso, aggiorna nel directory le informazioni relativa alla dimensione ed alle date, e libera la memoria occupata dalla tabella cui puntava `fp`.

Un'ultima osservazione, prima di chiudere. Errori nello specificare il nome del file sono tutt'altro che infrequenti, come ben puoi immaginare. Il guaio è che il sistema in un caso del genere ben si guarda dall'interromperti il programma: l'esecuzione continua, ma il file non è stato aperto e quindi devi aspettarti altri errori da qualche parte, chissà dove.[6] Però di dà un'informazione: ti restituisce in `fp` un codice di errore indicato con `NULL`.[7] Questo lo puoi verificare. Un buon consiglio è aprire i files con istruzioni di questo tipo:

```
FILE *fp;
if((fp=fopen("ciccio.mio","r")) == NULL) {
  perror("ciccio.mio");
  exit(1);
}
```

---

[6] Immagino i tuoi commenti, ma è giusto così: io potrei tentare di aprire un file e prevedere nel mio programma delle azioni di recupero se qualcosa non va.

[7] Non confondere i nomi `NUL` e `NULL`: il primo identifica un dato `char` il cui valore binario è zero; il secondo identifica l'indirizzo di memoria zero. Sono due cose ben diverse: pensa alla lunghezza del dato. Il compilatore *Gnu-CC* è un po' pedante, e bada bene a distinguerle.

La funzione `perror(...)` ti scrive sul terminale il nome del file (che le passi tu) e il motivo per cui non l'ha potuto aprire (ad esempio, il directory non esiste, o il disco è pieno). L'istruzione `exit(1)` termina il programma restituendo il valore `1` al sistema; questo viene interpretato come un codice di errore. Il sistema non se ne fa nulla, ma tu potresti usare questo codice,... se mai ti servirà imparerai come. Naturalmente, nessuno ti proibisce di sostituire la chiamata `exit(1)` con delle istruzioni che correggano la situazione di errore e permettano il proseguimento del programma.

6.2.5 Ed i files binari?

La sequenza di istruzioni è simile. L'apertura e la chiusura funzionano come per i files formattati, senza modifiche. Quelle che cambiano sono le istruzioni di lettura e scrittura. Eccole:

`fread(`⟨*buffer*⟩`, `⟨*cella*⟩`, `⟨*numero*⟩`, `⟨*file*⟩`);`
`fwrite(`⟨*buffer*⟩`, `⟨*cella*⟩`, `⟨*numero*⟩`, `⟨*file*⟩`);`

Qui le spiegazioni ci vogliono, ed in abbondanza.

- `fread` (file read) e `fwrite` (file write) sono, rispettivamente, le funzioni che eseguono la lettura e la scrittura di un blocco di dati.
- Il blocco di dati deve essere un'area contigua di memoria – ad esempio un vettore, o una singola variabile, destinato al trasferimento. Il campo ⟨*buffer*⟩ specifica l'indirizzo del dato o del blocco di dati da trasferire.
- La lunghezza del blocco di dati viene definita mediante due informazioni: la lunghezza della singola cella, ⟨*cella*⟩, ed il numero di celle, ⟨*numero*⟩. Il caso tipico è il trasferimento di un vettore, che supporrò per esempio essere di tipo `int`. In questo caso puoi specificare `sizeof(int)` per il campo ⟨*cella*⟩, e la dimensione del vettore per il campo ⟨*numero*⟩.
- Il file interessato dal trasferimento viene identificato dal puntatore che ti è stato restituito da `fopen`; quello che ho chiamato `fp`.
- Ambedue le funzioni restituiscono un valore `int` che contiene il numero di dati effettivamente trasferiti. Normalmente questo deve coincidere col numero di dati che hai specificato nella chiamata; se è diverso significa che si è verificato un errore. Ad esempio, hai tentato di leggere oltre la fine del file, nel qual caso ti viene restituito un numero di dati inferiore alle tue aspettative. Questo ti può essere utile per leggere files di cui non conosci a priori la lunghezza: quando non ci sono più dati smetti.

Lasciami insistere: i dati da trasferire devono essere contigui in memoria. Ricorda ancora una volta che la funzione, `fread` o `fwrite` che sia, riceve come informazioni solo l'indirizzo da cui iniziare il trasferimento e il numero di bytes da trasferire (che si calcola come prodotto degli argomenti ⟨*cella*⟩ e ⟨*numero*⟩). Nient'altro.

La scrittura o lettura mediante queste funzioni avviene in modo sequenziale:

- il sistema mantiene un puntatore all'interno del file; la posizione iniziale dopo la `fopen` è sul primo byte del file per l'accesso `"r"` o `"w"`, e dopo

l'ultimo byte per l'accesso `"a"`;
- ogni operazione di lettura o scrittura inizia dalla posizione corrente del puntatore;
- alla fine dell'operazione il puntatore è riposizionato dopo l'ultimo byte letto o scritto.

Va da sé che i files binari verranno tipicamente letti da un programma. Non che tu non possa esaminarne il contenuto binario: esistono dei programmini che ti permettono tranquillamente di stamparlo in decimale, ottale o esadecimale, a bytes o a word o anche in ASCII. Ma solitamente non è il massimo del divertimento.

La lettura dei dati deve essere fatta esattamente nell'ordine in cui sono stati scritti, rispettando le dimensioni. Ricorda che né le funzioni `fread` e `fwrite` né i vari personaggi che entrano in gioco si interessano del contenuto dei bytes che trasferisci o del nome che hai assegnato ai dati: la sola cosa che esiste sul file è una serie di bytes. La gestione spetta tutta e solo a te. Se ti capitasse di cambiare l'ordine di lettura rispetto a quello di scrittura, o di saltare la lettura di qualche dato, non ti resterebbe che piangere su te stesso e cercare l'errore.

6.2.6 L'accesso casuale ai dati su file

Quello che sto per dirti vale solo per quei files che ammettono un accesso casuale — nella stragrande maggioranza dei casi i dischi o assimilati. L'accesso sequenziale ai dati è utile ogniqualvolta si debbano elaborare in massa tutte le informazioni contenute nel file. Ma vi sono applicazioni che richiedono l'accesso a dati in modo selettivo, o comunque in un ordine diverso da quello di memorizzazione. Anche questo si può fare, naturalmente (ne dubitavi?).

Le funzioni disponibili sono fondamentalmente due:

```
long int ftell( FILE *fp);
int fseek( FILE *fp, long offset, int da_dove);
```

La funzione `ftell` restituisce la posizione corrente all'interno del file. È utile per memorizzare la posizione di un gruppo di dati che dovranno essere ritrovati in qualunque momento. Il valore restituito è di tipo `long int`, e dà la posizione in bytes a partire dall'inizio del file.

La funzione `fseek` forza il riposizionamento del puntatore interno al file associato a `fp`. Il secondo argomento, `offset`, specifica di quanto si deve traslare il puntatore; il terzo argomento, `da_dove`, specifica da dove deve essere calcolata la traslazione. Potrai specificare uno dei tre valori simbolici `SEEK_SET` (rispetto all'inizio del file), `SEEK_CUR` (rispetto alla posizione corrente) o `SEEK_END` (rispetto alla fine del file). Ad esempio, l'istruzione

```
fseek(fp,0L,SEEK_SET);
```

riposiziona il puntatore all'inizio, permettendo di rileggere (o riscrivere) il file daccapo. Se ti punge la curiosità di sapere che valore è assegnato a `SEEK_SET` prova a dare un'occhiata al file `stdio.h`.

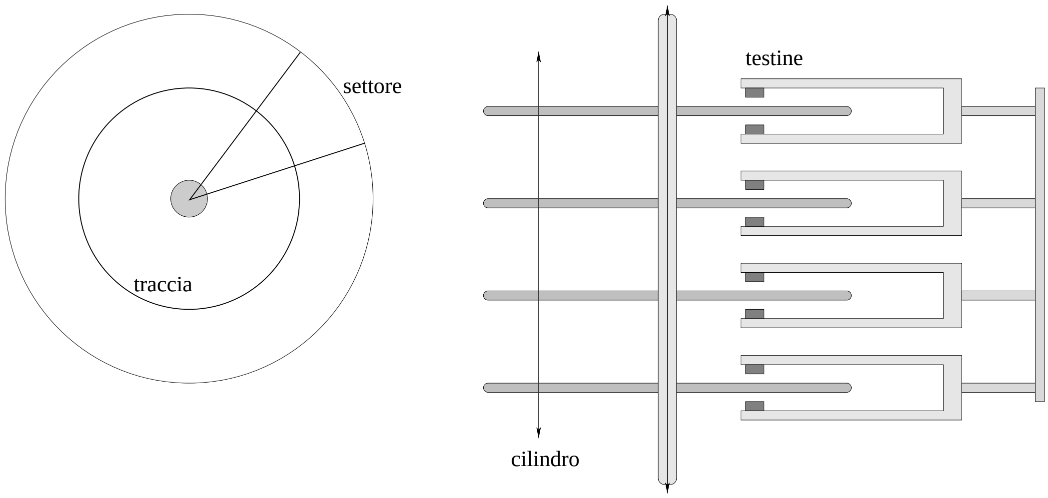

Figura 6.1. La struttura dei dischi.

6.3 Digressione: una sbirciatina ai dischi

Direi che è venuto il momento di guardare un po' dietro lo sportello del disco e spiegarti come il sistema ci possa mettere tanti files. Questo paragrafo non ha molto a che vedere con la programmazione in C; dovrebbe piuttosto aiutarti a meglio comprendere in che ambiente stai lavorando.

In questo paragrafo farò esplicito riferimento alla struttura dei PC, in particolare per quanto riguarda la descrizione del partizionamento e del meccanismo di boot.

6.3.1 La struttura fisica di un disco

Lo schema della struttura di un disco è rappresentato in figura 6.1. Il supporto fisico dell'informazione è un piatto metallico (disco rigido, o hard disk) o un supporto flessibile tagliato in forma circolare (dischetto flessibile, o floppy disk), ricoperti da uno strato magnetizzabile. La memorizzazione delle informazioni occupa delle circonferenze concentriche, dette *tracce*, o *tracks*, che vengono suddivise in *settori*, o *sectors*, come illustrato nella parte sinistra della figura 6.1. Nella maggior parte dei casi vengono utilizzate ambedue le facce di un disco. Inoltre le unità a dischi rigidi sono tipicamente dotate di più piatti, ciascuno dei quali è accessibile ad una coppia di *testine* (heads), come illustrato in figura 6.1. Le testine si muovono solitamente in blocco, in senso radiale, in modo da essere tutte alla stessa distanza dall'asse di rotazione del disco. La posizione del blocco di testine consente l'accesso ad un gruppo di tracce, tante quante sono le facce utilizzate, alle quali si dà il nome di *cilindro* (cylinder).

Il blocco di informazioni che può essere trasferito in una singola operazione di lettura o scrittura è quello che occupa la parte di traccia contenuta in un settore. Si parla, per l'appunto, di *blocco*. Tipicamente un blocco contiene

512 bytes. La terna di informazioni {*numero di testine, numero di cilindri, numero di settori*} definisce la cosiddetta *geometria* del disco. Ciascun blocco è identificato in modo univoco da tre numeri che specificano testina, cilindro e settore. Il numero totale di blocchi disponibili è determinato dalla geometria, e la capacità del disco è determinata dal numero di blocchi moltiplicato per la dimensione in bytes del blocco. [8]

Una caratteristica non trascurabile dei dischi è il *tempo di accesso*, ossia il tempo che trascorre mediamente tra la richiesta di informazione da parte della CPU e la disponibilità effettiva delle informazioni. I parametri rilevanti sono due:

i. *seek time*, o tempo di posizionamento delle testine, che devono spostarsi radialmente fino a raggiungere il cilindro richiesto;

ii. *latency time* o tempo di latenza: il tempo necessario perché il movimento rotatorio del disco porti il settore voluto a passare sotto le testine.

Naturalmente, il tempo di accesso è ben lungi dall'essere costante; i dati che trovi sulla documentazione tecnica dei dischi hanno carattere statistico, e rappresentano solo dei tempi medi.

La struttura qui descritta è comune alla maggior parte dei dischi, sia flessibili che rigidi. La differenza sta solo nella capacità del disco: si va da meno di 2 Megabytes per i dischetti flessibili (floppy) al centinaio di Gigabytes dei dischi rigidi di maggiore capacità (allo stato corrente, ma l'incremento è continuo). Fanno eccezione i CD-Rom, sui quali la scrittura avviene su un'unica traccia che procede a spirale dall'interno verso l'esterno, e che sono privi di una struttura a settori perché la densità di informazioni viene mantenuta costante lungo la traccia. In altre parole, la quantità di informazioni trasferite mentre il disco compie una singola rotazione varia gradualmente procedendo dalla parte interna a quella esterna del disco.

6.3.2 La formattazione e le partizioni

Una volta preparato il supporto fisico occorre costruire sul disco la struttura dei dati. Il livello più basso, solitamente invisibile all'utente finale, è la cosiddetta *formattazione a basso livello*. Si tratta, in termini semplificati, di un'operazione sistematica di scrittura di tutta la superficie del disco in cui vengono registrate all'inizio di ciascun settore delle informazioni di

---

[8] A voler ben guardare, tutto questo non corrisponde completamente alla verità. In termini più precisi, *era* vero tempo fa. Attualmente la geometria del disco dichiarata dal costruttore ed utilizzata per configurare i dischi fa ancora uso delle informazioni descritte, ma quella reale è più complessa: ad esempio, il numero di settori per traccia varia procedendo dalla periferia verso il centro del disco. Ciò consente di sfruttare al meglio la maggiore lunghezza delle tracce esterne per memorizzare un maggior numero di informazioni. La traduzione della terna (testina, cilindro, settore) fornita dal driver nel settore effettivo del disco è compito dell'elettronica di controllo installata sull'unità.

riferimento che serviranno poi per individuare il settore che interessa.

I dischi rigidi sul mercato sono già predisposti in fabbrica, e quindi la formattazione a basso livello non dovrebbe essere eseguita, di norma, dall'utente finale. A volte la si esegue per recuperare dei dischi che per qualche ragione l'hanno persa — o quando si sospetta che ciò sia avvenuto. Ma si tratta molto spesso di casi in cui dopo la formattazione del disco si perde anche la speranza di riutilizzarlo. Questa operazione era invece comune sui dischetti floppy, che venivano tipicamente messi in commercio non formattati. A questo provvedevano dei programmi distribuiti col sistema operativo. Ad esempio, il sistema *DOS* (Disk Operating System) ha un comando `format` che sui dischetti esegue sia la formattazione a basso livello che la preparazione del file system, di cui ti parlerò più avanti. Nel sistema Linux è ormai distribuito come standard un programma `superformat`. Attualmente anche i dischi floppy in commercio sono preformattati a basso livello.

La seconda operazione è la creazione delle *partizioni* del disco. Si tratta di una suddivisione logica del disco – non coinvolge il disco a livello fisico – in sezioni distinte che all'utente finale appaiono a tutti gli effetti come dischi distinti. Naturalmente si tratta di un'operazione che ha senso solo sui dischi di una certa capacità, tipicamente dischi rigidi; non si usa sui dischi floppy. Ecco, in breve, cosa avviene in pratica.

- Il primo settore del disco – da intendersi come il settore che ha numeri di testina, cilindro e settore zero – ha una funzione particolare: contiene tutte le informazioni da cui il software parte per ricostruire tutta la struttura logica del disco; a questo settore viene dato il nome *MBR (Master Boot Record).*
- Nel settore MBR c'è spazio per una tabella che contiene le informazioni sulle partizioni del disco. Normalmente i dischi posti in commercio hanno una sola partizione predefinita che occupa tutto il disco. Nella tabellina c'è spazio per un massimo di 4 partizioni.
- La struttura di partizioni esistente può essere modificata mediante programmi forniti con tutti i sistemi operativi. Ad esempio, sia DOS che Linux mettono a disposizione un programma `fdisk` destinato a questo scopo — il nome è lo stesso, ma il programma no.

La suddivisione in partizioni viene normalmente decisa prima di installare sul disco un sistema operativo. In questa fase si possono creare:

- fino a 4 partizioni *primarie*, oppure
- fino a 3 partizioni *primarie* ed una partizione *estesa*. La partizione estesa dovrà a sua volta essere suddivisa in partizioni *logiche*.

Tutto ciò può apparire bizzarro, soprattutto se si tien conto del fatto che ai fini dell'utilizzo – fatta salva una precisazione che ti dirò nel prossimo paragrafo – le partizioni primarie e logiche si comportano praticamente allo stesso modo. Il motivo sta semplicemente nel limite di 4 partizioni imposto dalla struttura del settore MBR: in 512 byte non c'è posto per molta roba! Se voglio creare più di 4 partizioni ci vuole un qualche trucco, e questo è costituito proprio dalle partizioni logiche: nel settore MBR c'è l'indicazione del punto di inizio della partizione estesa; la prima partizione logica parte dal primo settore della partizione estesa; in questo settore c'è l'indicazione del punto di partenza della seconda partizione logica, e così via, formando una catena.

È d'obbligo precisare che la struttura delle partizioni, in linea di principio, è piuttosto rigida. Di solito i manuali su questo punto hanno un atteggiamento un po' intimidatorio: una volta installato il software, le partizioni non si possono più modificare, pena la perdita di tutte le informazioni che si trovano sul disco.

In effetti, questo era vero quando non si disponeva di programmi abbastanza sofisticati da saper modificare la struttura delle partizioni in modo consistente. Anzi, si dovrebbe fare una rettifica: veniva perso il contenuto di tutte le partizioni effettivamente modificate. Oggi tutto questo resta vero se si usano programmi efficaci, ma non particolarmente sofisticati, come `fdisk`.[9] Ma esistono ormai normalmente in commercio programmi che sono in grado di modificare la struttura delle partizioni in modo consistente, senza perdita di informazioni. Meglio comunque mantenere un po' di diffidenza: personalmente, non ho mai usato un programma del genere senza aver provveduto a fare una copia di riserva di tutto quanto mi interessava conservare. Un minimo errore nel partizionamento potrebbe essere fatale.

6.3.3 Il file system

Una volta creata la struttura logica delle partizioni il disco, pur unico come periferico fisico, è visto come spezzato in più parti distinte, che vengono trattate di fatto come dischi separati. L'ultimo passo prima di procedere all'installazione di un sistema operativo è la creazione del *file system*. Anche questa è una struttura logica, ma mentre quella delle partizioni è identica su tutti i PC quella creata dal file system dipende dal sistema operativo, e talvolta lo stesso sistema operativo è in grado di gestire diverse strutture, tra loro diverse. Proprio per questo motivo non farò riferimento ad una struttura specifica, ma cercherò di mettere in evidenza quali siano gli elementi principali che, in un modo o nell'altro, sono presenti in qualunque file system.

Il fine primario di tutta la struttura, come ho già avuto modo di spiegarti nel paragrafo 1.2.2, è mettere l'utente finale in condizioni tali da poter accedere al contenuto di un file semplicemente specificandone il nome, ed al tempo stesso consentire all'utente di organizzare i propri files in un sistema di cartelle, o *directories*, che gli permettano di raggruppare i files che fanno riferimento a diverse attività. Un po' come nell'organizzazione di uno

[9] Non vorrei essere frainteso: `fdisk` è e resta uno strumento validissimo per creare la struttura iniziale delle partizioni o per intervenire quando il contenuto di un disco va comunque interamente riscritto. Si deve usare invece con molta cautela se ci sono sul disco informazioni che dovrebbero essere mantenute.

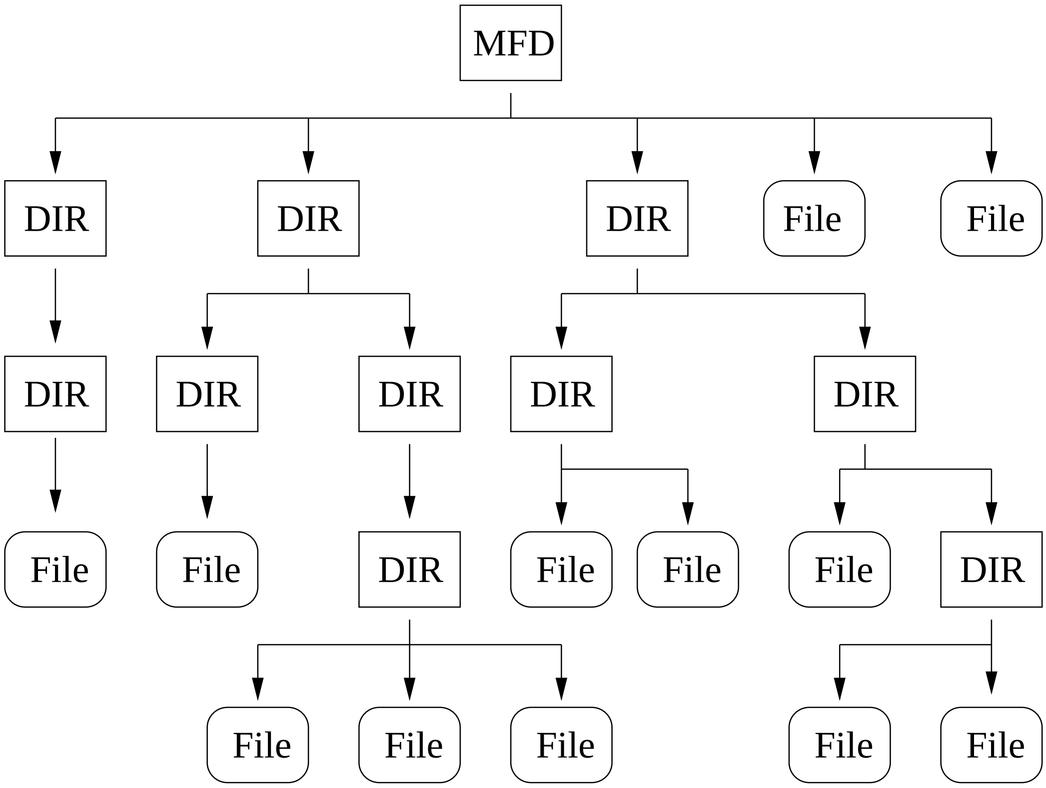

Figura 6.2. La struttura ad albero dei directories entro una partizione di un disco. Il punto di partenza della struttura è il Master File Directory (MFD).

schedario o di una biblioteca, ove diversi scaffali ospitano documenti o libri classificati secondo categorie logiche distinte, ed il tutto viene elencato in un catalogo generale. Alcuni sistemi – tipicamente quelli che possono gestire contemporaneamente l'attività di persone diverse, prevedono anche meccanismi di protezione reciproca che può andare dalla semplice garanzia che un comando accidentale di uno degli utenti non possa distruggere o modificare files altrui, alla protezione da qualunque tipo di accesso anche in sola lettura.

La struttura logica, per la parte direttamente visibile all'utente finale, è illustrata in figura 6.2. Ogni partizione ha un *Master File Directory (MFD)* che fa da radice di una struttura ad albero. Si tratta in pratica di un elenco di nomi a ciascuno dei quali corrisponde un *file* contenente dei dati. Per inciso, il master file directory stesso è un file che ha il solo privilegio di essere il punto di partenza di tutta la struttura. La posizione del MFD sul disco è memorizzata, in modo diretto o indiretto, in un punto prefissato: alcuni sistemi lo pongono in un blocco che si trova sempre nella stessa posizione rispetto all'inizio della partizione (ad esempio il secondo); altri sistemi memorizzano il suo indirizzo nel primo blocco della partizione, in una posizione fissata.

I files contenuti nel MFD possono essere files di dati (indicati in figura 6.2

con la scritta File) o essere a loro volta dei directories, indicati con DIR. Spesso ciascun directory contiene come nome anche se stesso ed il directory che lo precede nella struttura ad albero — detto anche *parent directory*. Il contenuto di un file di dati non riveste alcuna importanza per il file system. Ne ha invece il contenuto dei directories, perché, come per il MFD, è a sua volta un elenco di files dati o directories. Ne risulta una struttura gerarchica di cartelle, o contenitori, o, appunto, directories. Un po' come un edificio, che ha dei piani, che comprendono degli appartamenti, che sono suddivisi in stanze, che contengono dei mobili, che hanno dei cassetti o dei ripiani, che ospitano degli oggetti.

Un file entro la partizione viene identificato da un `path`, o cammino, che parte dal MFD, e percorre tutta la catena dei directories fino a quello che contiene il file, e finisce col nome del file stesso. In linea di principio occorrerebbe indicare il cammino completo ogni volta che si accede al file — un compito a dir poco sgradevole. Proprio per questo tutti i sistemi hanno qualche modo per definire un directory di lavoro corrente (detto anche *default directory*) dove cercare i files in mancanza di indicazioni più precise. Ma di questo dovresti essere già al corrente.

Tutto questo è quanto puoi vedere in modo diretto con in vari comandi che ti permettono di muoverti entro i directories ed elencarne il contenuto. Ma cosa c'è dietro le quinte?

Il fatto è che il disco – o la partizione – è solo una successione di settori o blocchi logici numerati a partire da 0. [10] Ed i bit che si trovano su disco, esattamente come quelli della memoria, non hanno colori diversi. Come si fa a stabilire quali blocchi appartengono ad un determinato file? Cerco di risponderti.

Qualunque file system deve mantenere:

- una tabella che tiene traccia dello stato dei singoli blocchi del disco, e precisamente se siano occupati da qualche file o disponibili;
- per ciascun file, una tabella che contenga l'elenco ordinato dei blocchi che appartengono a quel file: viene talvolta chiamata la *mappa* del file sul disco;
- per ciascun nome di file un rinvio alla mappa.

Le informazioni che ti ho elencato costituiscono il minimo indispensabile per una gestione efficace a consistente della struttura dei files. A queste si debbono aggiungere altre informazioni che, pur non essendo strettamente indispensabili per la gestione dello spazio su disco, sono tuttavia utili per il proprietario, quali ad esempio:

- la data ed ora di creazione del file;
- la data ed ora dell'ultima modifica apportata al file;

[10] Come ti ho spiegato, la traduzione del numero di blocco nella terna (testina,cilindro,settore) la fa il driver, e la traduzione successiva nel settore fisico effettivo la fa l'elettronica di controllo del disco.

- ◦ la data e l'ora dell'ultima copia di riserva del contenuto del file (ammesso che sia stata fatta);
- ◦ il proprietario del file;
- ◦ la dimensione del file, tipicamente in bytes;
- ◦ dei codici di protezione del file che ne impediscano la lettura o modifica o cancellazione accidentale da parte di utenti diversi dal proprietario.

L'elenco qui sopra è solo indicativo: le scelte operate da chi progetta i diversi file system possono essere le più varie. Ma occupiamoci più direttamente delle informazioni relative alla gestione dello spazio su disco.

La prima operazione comune a tutti i file systems è la suddivisione del disco in unità di allocazione. Idealmente questa dovrebbe coincidere con il blocco del disco, ossia la quantità minima di informazioni che possa essere letta o scritta in una singola operazione; tuttavia si preferisce spesso usare come unità di allocazione un gruppetto di blocchi contigui, detto *cluster*, di dimensione prefissata. [11]

La seconda operazione consiste nel preparare una tabella che associa a ciascun cluster una qualche informazione di stato. Esistono almeno due possibilità:

(1) ad ogni cluster si associa un bit: 1 significa che il cluster è riservato ad un file (quale, poco importa a questo livello), 0 significa che il cluster è disponibile; il complesso di queste informazioni viene detto *bitmap*, o mappa di bit;

(2) ad ogni cluster si associa un contatore che può avere come valore:
  - un numero che identifica il cluster successivo che appartiene allo stesso file;
  - un codice che identifica il blocco come l'ultimo di un file;

---

[11] Il motivo è apparentemente semplice: ogni unità di allocazione necessita di un certo spazio per la gestione, e questo viene sottratto allo spazio effettivamente disponibile; dunque, aumentando la dimensione del cluster si riduce lo spazio occupato per la gestione. A questa considerazione si aggiunge il fatto che memorizzare le informazioni di un file su un'area contigua di disco migliora l'efficienza delle operazioni di lettura e scrittura, perché si riducono i tempi morti dovuti al posizionamento delle testine. Ma c'è anche il rovescio della medaglia: i dati memorizzati su un file raramente riempiono in modo completo i cluster allocati. In coda ai dati resta sempre dello spazio inutilizzato, la cui dimensione aumenta di pari passo con quella del cluster. Per questo motivo i sistemi più sofisticati consentono di decidere entro ampi margini la dimensione del cluster al momento della creazione del file system. In termini rozzi, si tratta di prevedere se il disco dovrà ospitare prevalentemente files di grosse dimensioni – nel qual caso converrà aumentare la dimensione del cluster per migliorare l'efficienza del trasferimento di dati – o molti files piccoli – nel qual caso converrà privilegiare il risparmio di spazio inutilizzato scegliendo cluster piccoli. Fortunatamente per i meno esperti ogni sistema fa uso di valori di default che sono statisticamente validi per gli usi più comuni.

- un codice che identifica il blocco come disponibile;
- un codice che identifica il blocco come non utilizzabile, per qualche ragione.

Il modo in cui viene memorizzata la mappa di occupazione del file dipende in modo abbastanza stretto da quanto ho detto qui sopra.

◦ Se l'occupazione dei cluster viene gestita con la tecnica della bitmap, allora occorre una tabella ausiliaria che contiene l'elenco ordinato dei blocchi che appartengono al file. L'indirizzo ove trovare questa tabella deve trovarsi nel directory, accanto al nome del file.

◦ Se al cluster viene associato un contatore allora il meccanismo è, almeno apparentemente, più semplice: nel directory si trova l'indirizzo del primo blocco; il contatore associato al primo blocco dice qual è il successivo &c, fino al contatore associato all'ultimo blocco che contiene il codice che lo identifica come tale.

Per quanto sommarie, le informazioni che ti ho dato descrivono in modo sufficientemente completo i vari modi di gestione dello spazio su disco. Esaminiamo, ad esempio, il caso della bitmap.

◦ Al momento della creazione del file system viene allocato lo spazio per la bitmap: un bit per ciascun cluster, in una zona del disco il cui indirizzo e dimensione si trovano in una posizione prefissata.

◦ Nello stesso momento viene creata la struttura del MFD, nel quale vengono inseriti:
  - il nome del MFD stesso;
  - l'indirizzo del blocco che contiene la mappa di occupazione del file che costituisce il MFD;
  - l'indirizzo del MFD su disco.

  Queste informazioni permettono di ricostruire il contenuto del MFD.

◦ Sempre nello stesso momento, vengono dichiarati occupati nella bitmap tutti i cluster occupati dalla bitmap stessa e dal MFD.

◦ La creazione di un nuovo file – che potrebbe essere a sua volta un directory – da inserire nel MFD richiede:
  - la ricerca nella bitmap di un certo numero di cluster contrassegnati con 0, e quindi liberi, destinati a contenere la mappa di occupazione ed i dati del file stesso; i cluster vengono dichiarati occupati nella bitmap;
  - la scrittura della mappa di occupazione del file;
  - la scrittura dei dati, blocco per blocco, mantenendo l'ordine della mappa di occupazione.

◦ La creazione di un file da inserire in un directory diverso dal MFD richiede praticamente le stesse operazioni, mutatis mutandis.

◦ La cancellazione di un file viene eseguita rovesciando la procedura:
  - i cluster elencati nella mappa di occupazione vengono dichiarati liberi, così come quello che contiene la mappa di occupazione;

- il nome viene rimosso dal directory, assieme a tutte le informazioni ad esso associate.

Voglio sottolineare ancora una volta che lo schema illustrato qui è solo un esempio sommario: la progettazione di un file system comprende spesso dei meccanismi di controllo che consentano di recuperare gli errori causati, ad esempio, da un blocco improvviso del sistema o da un'improvvisa mancanza di corrente — causata magari dalla fretta dell'operatore che ha spento la macchina in modo non ordinato. Ma non credo sia necessario aggiungere di più: se vuoi conoscere in dettaglio la struttura dei files sul tuo sistema non hai che da cercare la documentazione adatta.

6.3.4 Il meccanismo di boot

E veniamo a uno dei punti solitamente misteriosi: come fa il sistema a partire? L'operazione coinvolge diversi personaggi: il BIOS di sistema, il settore MBR, la partizione attiva, il kernel del sistema. Andiamo con ordine.

Il BIOS (Basic I/O System) è un blocchetto di memoria ROM (Read Only Memory) installato fisicamente sulla scheda madre del sistema, che contiene dei programmi e dati memorizzati in modo permanente. [12] Tra ai programmi ce ne interessano due:

- il POST (Power–On Self Test), che esegue una serie di test di buon funzionamento della macchina;
- il programma di caricamento del sistema operativo, che viene lanciato subito dopo l'esecuzione del POST.

Ad attivare il POST ci pensa l'elettronica, è un problema di chi progetta la scheda, che in pratica deve fare in modo che all'accensione venga caricato nella CPU l'indirizzo dell prima istruzione del POST, e poi la CPU inizi a lavorare. [13]

Eseguiti i test, comincia l'avvio vero e proprio. Questa operazione viene chiamata *bootstrap*, solitamente abbreviato in *boot*. [14] Nella sua forma più semplice la sequenza delle operazioni è questa:

---

[12] Permanente,... ormai praticamente tutte le macchine dispongono di una memoria BIOS riscrivibile, ed i produttori forniscono programmi che provvedono alla riscrittura in caso di aggiornamento. Il fatto è che ormai i programmi BIOS sono diventati piuttosto complessi, e come tali soggetti ad errori che richiedono degli aggiornamenti.

[13] In effetti, sui vecchi calcolatori l'avvio del sistema dopo l'accensione veniva provocato manualmente, usando una serie di pulsanti o interruttori presenti sul pannello di controllo, proprio caricando l'indirizzo di una memoria ROM e premendo il pulsante START. A volte le istruzioni di avvio venivano addirittura caricate a mano, direttamente in binario.

[14] Il termine bootstrap, in senso letterale, è la linguetta che serve per calzare lo stivale. Viene anche usato per indicare un'attività che viene iniziata col minimo aiuto esterno.

- il programma di caricamento del BIOS legge il primo blocco del disco copiandolo in memoria a partire da un indirizzo prefissato. Il primo blocco contiene, oltre al resto, anche le istruzioni per il caricamento di qualcos'altro. Finita la lettura il BIOS manda in esecuzione le istruzioni che ha appena caricato.
- il programmino che sta nel primo blocco cerca nella tabella delle partizioni – che ormai è in memoria – l'indicatore della partizione attiva.
- il primo settore della partizione attiva contiene a sua volta un programma che è stato predisposto per il caricamento della prima parte del kernel a partire da un blocco fissato di disco. Il programmino lo porta in memoria e gli cede il controllo.
- la prima parte del kernel provvede a fare il resto.

La partizione attiva deve essere una partizione primaria: è questa la piccola differenza tra partizioni primarie e partizioni logiche che ti avevo preannunciato. Il motivo è semplice: il programmino che si trova sul blocco di boot non prevede solitamente la ricerca sulle partizioni logiche.

Lo schema che ti ho illustrato, come puoi ben osservare, prevede che sul disco esista una partizione predefinita come attiva che contenga sia un sistema operativo da caricare sia, nel primo settore della partizione, un programma di caricamento. La partizione attiva viene selezionata col programma `fdisk` durante la creazione delle partizioni; alla scrittura del programmino che carica il resto provvede la procedura di installazione del sistema. [15] Va da sé che in questo schema non c'è spazio per l'installazione di sistemi diversi.

La possibilità di caricare sistemi diversi esiste in quanto il programmino che si trova sul settore MBR può essere modificato. I modi possibili sono due: si dà la possibilità di decidere al momento dell'accensione quale partizione attivare, battendone il numero sulla tastiera, oppure – più frequentemente – si provoca direttamente il caricamento di un programma un po' più complesso che si inserisce prima della lettura del primo blocco della partizione attiva. Questo programma, detto *boot loader*, è in grado di stabilire un dialogo con l'utente finale, ad esempio presentandogli le scelte possibili ed attendendo un comando. Viene realizzata in questo modo la possibilità di installare sistemi diversi su partizioni distinte, e di decidere al momento dell'accensione quale di questi avviare.

---

[15] Se ti sei divertito un po' col sistema DOS saprai che durante la formattazione puoi scegliere l'opzione di installare il sistema e rendere il disco avviabile: è proprio questa opzione che scrive il kernel del sistema da qualche parte nella partizione, e scrive sul primo blocco il programmino che carica il resto.