**Introduzione**

Tutte le applicazioni per computer devono memorizzare e recuperare informazioni. Durante la sua esecuzione un processo può salvare all’interno del proprio spazio degli indirizzi un numero limitato di informazioni. La capacità di memorizzazione è in ogni caso limitata dalla dimensione dello spazio degli indirizzi virtuali. Per alcuni tipi di applicazioni una certa dimensione può essere adeguata, ma può rivelarsi troppo piccola per altri.

Un secondo problema relativo al mantenimento di queste informazioni all’interno dello spazio degli indirizzi di un processo è che quando il processo finisce, l’informazione va persa. Per molte applicazioni (per esempio i database) le informazioni devono essere conservate per settimane, mesi o anche indefinitamente. Non è accettabile che svaniscano nel momento in cui il processo che le usa termina. Inoltre esse non devono andare perse quando un problema del computer causa inaspettatamente la fine del processo.

Un terzo aspetto problematico si verifica con la necessità di accedere alle informazioni

(o a parte di esse) nel medesimo istante da parte di processi multipli. Nel caso di una rubrica telefonica online all’interno dello spazio degli indirizzi di un singolo processo, solo quel dato processo può accedervi. Una possibile soluzione è rendere questa stessa informazione indipendente da qualunque processo.

I requisiti essenziali per il salvataggio delle informazioni a lungo termine sono quindi tre:

1. deve essere possibile salvare una grandissima quantità di informazioni;

2. le informazioni devono permanere oltre la fine del processo che le usa;

3. le informazioni devono poter essere acquisite contemporaneamente da molteplici processi.

Per questo tipo di memorizzazione a lungo termine sono stati usati per anni i dischi magnetici, oltre ai nastri e ai dischi ottici, che però hanno prestazioni molto inferiori.

Un disco contiene una sequenza lineare di blocchi dalle dimensioni fisse e supporta due operazioni:

1. lettura del blocco k;

2. scrittura del blocco k.

Si tratta di operazioni molto scomode, specialmente su grossi sistemi usati da molte applicazioni ed eventualmente da diversi utenti concorrenti.

Alcune delle tante domande che possono sorgere sono le seguenti.

1. Come trovare le informazioni?

2. Come evitare che un utente legga le informazioni di un altro?

3. Come sapere quali blocchi sono liberi?

Proprio come abbiamo visto che un sistema operativo astrae il concetto di processore per creare l’astrazione di processo e che astrae il concetto di memoria fisica per offrire ai processi spazi di indirizzi (virtuali), anche in questo caso possiamo risolvere il problema con un’astrazione:

il file.

Le astrazioni di processo (e thread), di spazio degli indirizzi e di file sono i concetti principali dei sistemi operativi.

I file sono unità logiche di informazioni create dai processi. I processi possono leggere file esistenti e crearne di nuovi se necessario. Le informazioni salvate nei file sono persistenti, cioè non influenzate dalla creazione e dalla fine del processo. Un file dovrebbe scomparire solo quando il suo proprietario lo rimuove esplicitamente.

I file sono gestiti dal sistema operativo. Come sono strutturati, denominati e come accedervi, utilizzarli, proteggerli, implementarli e gestirli sono tra gli argomenti principali della progettazione dei sistemi operativi. La parte del sistema operativo che nella sua interezza ha a che fare con i file è conosciuta come file system.

Dal punto di vista dell’utente, l’aspetto principale di un file system è come esso appare, ossia, che cosa costituisca un file, come siano denominati e protetti i file, quali operazioni siano consentite sui file e così via. Il fatto che per tenere traccia della memoria libera siano usati elenchi collegati piuttosto che bitmap e quanti siano i settori all’interno di un blocco logico non sono aspetti importanti per l’utente, ma lo sono molto per i progettisti di file system. Analizzeremo entrambe le parti.

**I file**

In questa sezione studieremo i file dal punto di vista dell’utente.

**Nomi dei file**

La principale caratteristica di qualsiasi meccanismo di astrazione è il modo in cui sono denominati gli oggetti gestiti.

Quando un processo crea un file, dà a quel file un nome. Quando il processo termina, il file continua a esistere e altri processi possono avervi accesso, usando il suo nome. Le regole esatte per la denominazione dei file possono in qualche modo variare da sistema a sistema, ma tutti i sistemi operativi attuali consentono, come nomi validi stringhe che vanno da uno a otto caratteri, inclusi numeri e caratteri speciali, come ! o accentati (à è ì…) tranne quelli usati dal file system come:

~ “ # % & \* : < > ? \ / | { }...

Molti file system supportano nomi lunghi fino a 255 caratteri.

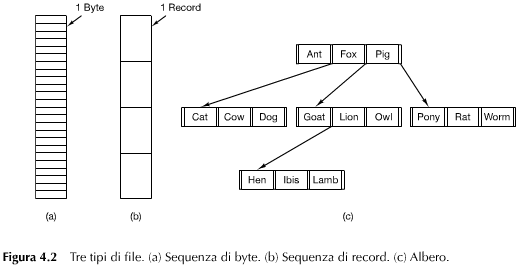
Alcuni file system distinguono fra lettere maiuscole e minuscole, mentre altri no. UNIX ricade nella prima categoria, MS-DOS nella seconda.

Molti sistemi operativi supportano nomi di file composti da due parti, separate da un punto, come nel caso di *prog.c*. La parte che segue il punto è detta estensione del file e indica generalmente un aspetto particolare del file.

In MS-DOS, per esempio, i nomi dei file vanno da 1 a 8 caratteri, più un’estensione opzionale da 1 a 3 caratteri.

**Struttura dei file**

I file possono essere strutturati in tanti modi diversi. Quelle più comuni sono descritte nella Figura 4.2.



Il file nella Figura 4.2(a) è una sequenza non strutturata di byte.

In effetti il sistema operativo non conosce e non è interessato a che cosa stia nel file. Tutto ciò che vede sono byte. Il significato deve essere imposto dai programmi a livello utente.

Il fatto che il sistema operativo consideri i file come sequenze di byte fornisce la massima flessibilità. I programmi utente possono mettere ciò che vogliono nei loro file e chiamarli nel modo più conveniente. Il sistema operativo non fornisce aiuto, ma allo stesso tempo non interviene. Tutte le versioni di UNIX, MS-DOS e Windows usano questo modello.

Una prima ipotesi di struttura è mostrata nella Figura 4.2(b).

In questo modello un file è una sequenza di record a lunghezza fissa, ciascuno con una certa struttura interna. Centrale, rispetto all’idea di un file composto da una sequenza di record, è che l’operazione di lettura restituisca un record e l’operazione di scrittura sovrascriva o aggiunga un record.

I primi mainframe utilizzavano record da 80 caratteri così come le 80 colonne di una scheda perforata.

Il terzo tipo di struttura dei file è mostrato nella Figura 4.2(c).

In questa organizzazione un file è composto da un albero di record, non necessariamente della stessa lunghezza, ognuno contenente un campo chiave in una posizione fissa del record. L’albero è ordinato sul campo chiave per permettere una ricerca rapida per una chiave particolare. L’operazione base in questo caso non è prendere il record “successivo”, sebbene sia comunque possibile, ma prendere il record con una chiave ben precisa.

**Tipi dei file**

Molti sistemi operativi supportano diversi tipi di file. UNIX e Windows per esempio hanno file e directory normali. UNIX ha anche file speciali a caratteri o a blocchi.

I file normali sono quelli contenenti informazioni utente. I file della Figura 4.2 sono file normali. Le directory sono file di sistema usati per mantenere la struttura del file system.

I file speciali a caratteri sono relativi all’input/output e usati per modellare i dispositivi seriali di I/O, come terminali, stampanti e network.

I file speciali a blocchi sono usati per modellare i dischi.

I file normali sono generalmente sia file ASCII sia file binari.

I file ASCII sono composti da linee di testo. In alcuni sistemi ciascuna linea termina con un carattere di “a capo” (carriage return). In altri è usato il carattere di “nuova riga” (line feed). Alcuni sistemi usano entrambi (per esempio Windows). Le linee non necessariamente sono della stessa lunghezza.

Il grande vantaggio dei file ASCII è che possono essere visualizzati e stampati così come sono e possono essere corretti con un qualunque editor di testo. Inoltre, i programmi che utilizzano file ASCII per l’input/output possono essere interfacciati con facilità (basta collegare l’output di un programma all’input di un altro).

Altri file sono binari, il che significa semplicemente che non sono file ASCII. Essi sono sequenze di bit che non vengono interpretati come caratteri e la loro struttura interna è conosciuta solo dai programmi preposti a usarli. Quindi, non possono essere stampati e maneggiati da editor di testi.

Per esempio, nella Figura 4.3(a) vediamo un semplice file binario eseguibile preso da una delle

prime versioni di UNIX.

Sebbene tecnicamente il file sia solo una sequenza di byte, il sistema operativo esegue il file solo se ha un formato corretto:

- intestazione (header);

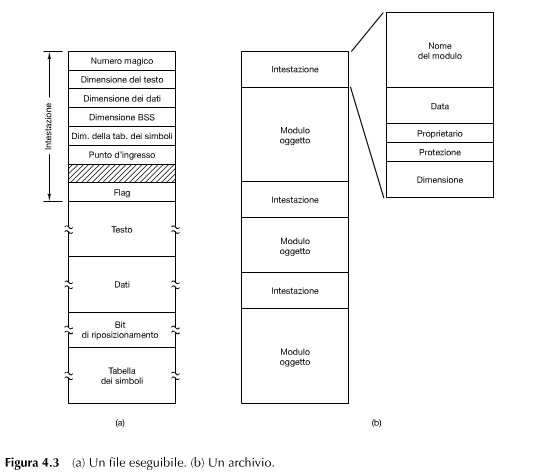
-comprende il numero magico → indica se il file sia eseguibile o no.

- testo;

- dati;

- bit di rilocazione (relocation bit);

- tabella dei simboli → abilita i processi di debug.



L’intestazione parte con un cosiddetto numero magico che identifica il file come eseguibile. Questo numero è seguito da:

- le dimensioni dei segmenti che compongono il file (testo, dati, BSS e tabella dei simboli);

- l’indirizzo di inizio del programma (entry point);

- alcuni flag.

**Accesso ai file**

I primi sistemi operativi avevano un solo tipo di accesso ai file: l’accesso sequenziale.

In questi sistemi un processo poteva leggere tutti i byte o i record in ordine, a partire dal principio, ma non poteva saltarli o leggerli in maniera disordinata. I file sequenziali erano perfetti quando l’archiviazione era su nastro magnetico piuttosto che su disco, visto che potevano essere riavvolti.

Con la comparsa dei dischi per la memorizzazione dei file fu possibile leggere i byte o i record di un file in maniera disordinata o accedere ai record secondo una chiave piuttosto che in base alla posizione. I file i cui byte o record possono essere letti in qualsiasi ordine sono chiamati file ad accesso casuale.

Per specificare dove cominciare a leggere possono essere usati due metodi.

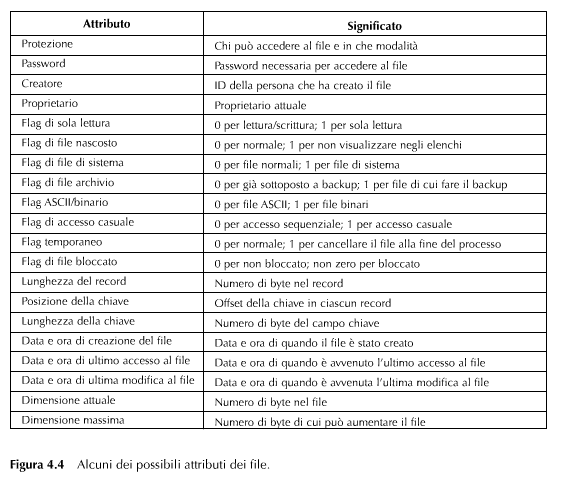
L’operazione *read()* permette di leggere i dati a partire da una posizione specifica nel file.

L’operazione *seek()* permette di impostare la posizione corrente nel file in modo da poterlo poi leggere sequenzialmente da quella posizione. Quest’ultimo metodo è usato sia in UNIX sia in Windows.

**Attributi dei file**

Ogni file ha un nome e i propri dati. Inoltre, tutti i sistemi operativi associano informazioni a ciascun file, per esempio la data e l’ora in cui il file è stato modificato l’ultima volta e la dimensione del file. Chiameremo attributi queste voci extra del file. Qualcuno li chiama metadati.

L’elenco degli attributi cambia in modo considerevole da sistema a sistema. La tabella della Figura 4.4 mostra le più comuni.



**Operazioni sui file**

Di seguito sono descritte le chiamate di sistema più comuni relative ai file.

1) *create*: crea un file vuoto con alcuni attributi impostati.

2) *delete*: rimuove il file e libera lo spazio sul disco.

3) *open*: prima di poter usare un file, un processo deve aprirlo. Lo scopo della chiamata *open()*

è di permettere al sistema di portare nella memoria principale gli attributi e l’elenco degli indirizzi del disco per un accesso rapido nelle chiamate a seguire.

4) *close*: chiude il file liberando la memoria dai riferimenti al file perché non verrà più utilizzato. Molti sistemi incoraggiano la cosa imponendo ai processi un numero massimo di file

aperti. Un disco è scritto a blocchi e la chiusura di un file forza la scrittura dell’ultimo blocco del file, anche se quel blocco potrebbe non essere ancora completamente pieno.

5) *read*: legge i dati dal file. Generalmente i byte vengono letti dalla posizione attuale.

Il chiamante deve specificare quanti dati sono necessari e deve anche fornire un buffer in cui metterli.

6) *write*: scrive i dati nel file, dalla posizione corrente. Se l’attuale posizione è la fine del file, la sua dimensione aumenta.

7) *append*: aggiunge dati alla fine del file.

8) *seek*: cambia la posizione corrente nei file ad accesso casuale. Dopo questa chiamata i dati possono essere letti o scritti a partire da quella posizione.

9) *get attributes*: permette di leggere gli attributi dei file.

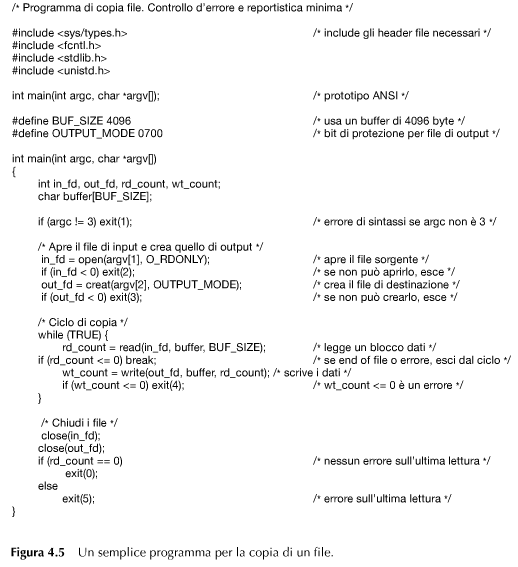
10) *set attributes*: permette all’utente di impostare alcuni degli attributi.

11) *rename*: permette di rinominare il file.

**Un esempio di programma che usa le chiamate del file system**

In questo paragrafo prenderemo in esame un semplice programma UNIX che copia un file

da uno sorgente a uno di destinazione. Il codice è riportato nella Figura 4.5.



Il programma ha funzionalità minimali e una ancor peggiore gestione degli errori, ma dà un’idea di come lavorino alcune chiamate di sistema relative ai file.

Il programma, *copyfile*, può essere per esempio chiamato dalla linea di comando

*copyfile abc xyz*

per copiare il file *abc* in *xyz*.

Se *xyz* esiste già, sarà sovrascritto, altrimenti sarà creato. Il programma deve essere chiamato esattamente con due parametri, entrambi due nomi di file consentiti. Il primo è il sorgente; il secondo il file di output.

Le quattro dichiarazioni di *#include* vicine all’inizio del programma causano l’inclusione nel programma di una gran quantità di definizioni e prototipi di funzioni. Queste sono necessarie per rendere il programma conforme agli standard internazionali pertinenti, ma non ci interessano ulteriormente.

La riga successiva è un prototipo di funzione per *main*, richiesto dall’ANSI C, ma anch’esso non significativo per i nostri scopi.

La prima istruzione *#define* è una definizione che dichiara la stringa di caratteri *BUF\_SIZE* come una macro che si traduce nel numero 4096. Il programma leggerà e scriverà in pezzi di 4096 byte.

La seconda dichiarazione *#define* determina chi può accedere al file di output.

Il programma principale è chiamato *main* e ha due parametri, *argc* e *argv*. Sono forniti dal sistema operativo quando viene chiamato il programma. Il primo indica quante stringhe sono presenti nella riga di comando che invoca il programma, incluso il nome programma. Dovrebbero essere 3.

Il secondo è un array di puntatori ai parametri. Tramite questo array il programma accede ai suoi parametri.

Sono dichiarate cinque variabili. Le prime due, *in\_fd* e *out\_fd*, conterranno i descrittori dei file, piccoli numeri interi restituiti quando il file viene aperto.

Le due successive, *rd\_count* e *wt\_count*, sono i contatori dei byte restituiti rispettivamente dalle chiamate di sistema *read* e *write*. L’ultima, *buffer*, è il buffer usato per contenere i dati letti e fornire i dati da scrivere.

La prima istruzione reale controlla *argc* per vedere se è uguale a 3. Se non lo è, esce con codice di stato 1. Ogni codice di stato diverso da 0 significa che vi è stato un errore. Il codice di stato è la sola segnalazione di errore in questo programma.

Quindi proviamo ad aprire il file sorgente e a creare il file di destinazione. Se il sorgente è aperto con successo, il sistema assegna a *in\_fd* un intero piccolo, per identificare il file. Le chiamate seguenti devono includere questo intero in modo che il sistema riconosca il file voluto. Allo stesso modo, se viene creato il file di destinazione, *out\_fd* è valorizzato per identificarlo. Il secondo argomento di *create* imposta la modalità di protezione.

Se l’apertura o la creazione falliscono, il descrittore del file corrispondente è impostato a -1 e il programma esce con un codice di errore.

Arriva adesso il ciclo di copia. Comincia cercando di leggere 4 KB di dati in *buffer*. Lo fa chiamando la procedura di libreria *read*, che invoca in effetti la chiamata di sistema *read*.

Il primo parametro identifica il file, il secondo dà il buffer e il terzo indica quanti byte leggere.

Il valore assegnato a *rd\_count* fornisce il numero di byte letti esattamente. Normalmente questo sarà 4096, a meno che nel file non siano rimasti meno byte. Quando viene raggiunta la fine del file, sarà 0. Se *rd\_count* è 0 o negativa, la copia non può proseguire e così è eseguita l’istruzione break per terminare il ciclo (altrimenti infinito).

La chiamata *write* scrive in output il buffer nel file di destinazione.

Il primo parametro identifica il file, il secondo fornisce il buffer e il terzo indica quanti byte scrivere. Si noti che il conteggio dei byte è il numero dei byte effettivamente letti e non *BUF\_SIZE*.

Quando l’intero file è stato processato, la prima chiamata oltre la fine file restituirà 0 in *wt\_count*, il che farà uscire dal ciclo.

A questo punto i due file sono chiusi e il programma esce con un codice di stato che ne indica la fine in condizioni normali.

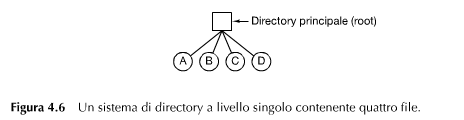
**Le directory**

Per tener traccia dei file, i file system normalmente hanno directory o cartelle, che in molti sistemi sono anch’esse dei file.

**Sistemi di directory a livello singolo**

La forma più semplice di sistema di directory è di avere una sola directory contenente tutti i file, talvolta chiamata directory principale (root directory).

Nella Figura 4.6 è riportato un esempio di sistema con una directory. In questo caso, la directory contiene quattro file.



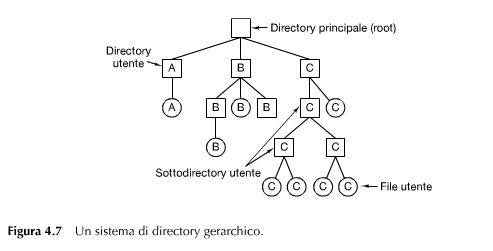
Il vantaggio di questo schema è la semplicità e la capacità di localizzare i file rapidamente, c’è solo un posto dove guardare.

**Sistemi di directory gerarchici**

Il singolo livello è adeguato a semplici applicazioni dedicate. Di conseguenza serve un altro modo.

Quello che serve è una gerarchia (cioè delle directory ramificate ad albero). Basta adottare una definizione ricorsiva in cui ogni directory può contenere altre directory (e/o file), con il vincolo che per ciascuna di esse esista sempre un solo contenitore (o genitore).

Questo metodo è illustrato nella Figura 4.7.



**Nome di percorso**

Quando il file system è organizzato secondo un albero di directory, i nomi dei file vanno specificati in qualche modo. Sono usati comunemente due metodi.

Nel primo metodo è assegnato a ogni file un nome di percorso assoluto composto dal percorso dalla directory principale al file. I nomi di percorso assoluti iniziano dalla directory principale e sono univoci.

Il percorso è composto dalla sequenza ordinata delle directory, separate da un carattere speciale “/” in UNIX, “\” in Windows e “>” in MULTICS.

Se il primo carattere del nome percorso è il separatore, allora il riferimento è assoluto.

L’altro metodo è quello del nome di percorso relativo. In questo metodo, il file o la directory viene individuato utilizzando un punto relativo sull’albero (la directory corrente, la home dell’utente, …).

La maggior parte dei sistemi operativi hanno in ciascuna directory due file speciali:

- “.” è la directory corrente;

- “..” è il suo genitore (tranne per root che fa riferimento a se stesso).

**Operazioni sulle directory**

1) *mkdir*: crea una directory vuota, ad eccezione di “.” e “..” messi in automatico.

2) *rmdir*: rimuove una directory. Per essere rimossa la directory deve essere vuote, cioè contenere al più “.” e “..”.

3) *opendir*: carica in memoria tutti i riferimenti di localizzazione sul disco prima della lettura (così come avviene per i file).

4) *closedir*: chiude la directory al termine della lettura e libera il corrispondente spazio di memoria.

5) *readdir*: restituisce la successiva voce di una directory aperta.

6) *rename*: rinomina una directory esistente.

7) *link*: il collegamento (linking) è una tecnica che permette a un file di apparire in più di una directory. Questa chiamata di sistema specifica un file esistente e un path name e crea un collegamento fra il file esistente e il nome specificato nel percorso. In questo modo il file appare in molteplici directory. Un collegamento di questo tipo, che aumenta il contatore nell’i-node file (per tener traccia del numero di directory contenenti il file) è detto hard link

(o collegamento fisico).

8) *unlink*: rimuove il collegamento nella directory, se il file è unico è cancellato dal file system altrimenti è rimosso solo il collegamento (viene scollegato, unlinked).

**Implementazione del file system**

È giunto il momento di spostarsi dalla visione utente del file system a quella dell’implementatore.

Agli utenti interessa come sono chiamati i file, che operazioni si possono fare su di loro, com’è fatto un albero di directory e simili questioni di interfaccia. Agli implementatori interessa il modo in cui sono memorizzati file e directory, com’è gestito lo spazio su disco e come fare affinché tutto funzioni con efficacia e affidabilità.

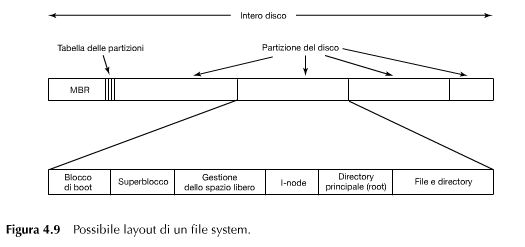
**Layout del file system**

I file system sono memorizzati sul disco. La maggior parte dei dischi è suddivisibile in una o più partizioni, con file system indipendenti su ciascuna.

Il settore 0 del disco è chiamato MBR (master boot record) ed è usato per avviare il computer.

La fine dell’MBR contiene la tabella delle partizioni. Questa tabella contiene gli indirizzi di inizio e di fine di ciascuna partizione. Una delle partizioni della tabella è indicata come attiva.

Anche se in generale il contenuto delle partizioni cambia da file system a file system, uno schema ricorrente è mostrato nella Figura 4.9.



Il superblocco contiene tutti i parametri chiave riguardanti il file system ed è letto in memoria all’avvio del computer o quando il file system viene usato per la prima volta. Le informazioni tipiche nel superblock includono un numero magico che identifica il tipo di file system, il numero di blocchi nel file system e altre informazioni amministrative chiave.

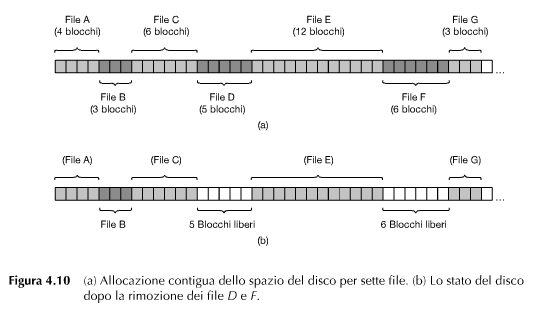
**Implementazione dei file**

Probabilmente l’aspetto principale relativo all’implementazione della memorizzazione dei file è il tener traccia di quali blocchi del disco siano associati a un determinato file. A seconda dei diversi sistemi operativi sono usati metodi diversi.

**Allocazione contigua**

Lo schema di allocazione più semplice è quello di memorizzare ciascun file come una sequenza contigua di blocchi del disco. In questo modo su un disco con blocchi da 1 KB, un file di 50 KB allocherebbe 50 blocchi consecutivi. Con blocchi di 2 KB ne allocherebbe 25 consecutivi.

Nella Figura 4.10(a) vediamo un esempio di allocazione contigua dello spazio.



Si noti che ogni file parte dall’inizio di un nuovo blocco, quindi se il file fosse realmente di 3 blocchi e ½, parte dello spazio sarebbe sprecato all’interno della fine dell’ultimo blocco.

Nella figura sono mostrati un totale di sette file, ognuno dei quali inizia dal blocco successivo che segue la fine del file precedente.

L’allocazione di spazio contiguo su disco presenta due vantaggi importanti. Innanzitutto è semplice da implementare poiché tener traccia della posizione dei blocchi di un file si riduce a due numeri: l’indirizzo del disco del primo blocco e il numero dei blocchi del file. Dato il numero del primo blocco, il numero di qualunque altro blocco si ricava con una semplice addizione.

In secondo luogo, poiché l’intero file può esser letto dal disco con una singola operazione le prestazioni in lettura sono eccellenti.

Sfortunatamente l’allocazione contigua presenta anche un rovescio della medaglia abbastanza significativo: nel corso degli anni i dischi si frammentano. Per vedere come ciò accada, esaminiamo la Figura 4.10(b). In questa figura vi sono due file, D e F, che sono stati rimossi. Quando viene eliminato un file vengono naturalmente liberati i suoi blocchi, lasciando un intervallo di blocchi liberi sul disco. Il disco non viene immediatamente compattato per chiudere il buco, visto che ciò potrebbe comportare la copia di tutti i blocchi che seguono il buco, potenzialmente milioni. Di conseguenza il disco risulta alla fine composto da file e buchi, come mostra la figura.

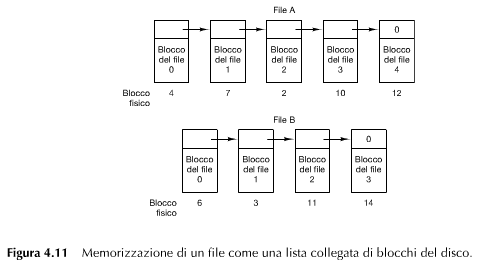
L’allocazione contigua è stata utilizzata inizialmente con i dischi magnetici in ragione alle loro caratteristiche. Il sistema è stato abbandonato perché all’atto della creazione del file era necessario specificarne la dimensione massima (non sempre nota in anticipo).

C’è tuttavia una situazione in cui l’allocazione contigua è, a tutti gli effetti, fattibile e largamente usata: sui CD-ROM. In questi casi si parla di “cicli e ricicli della storia dell’informatica” per indicare che anche con l’incremento tecnologico, in modo bizzarro e inaspettato, algoritmi o metodi divenuti ormai obsoleti ritornino ad essere attuali.

**Allocazione a liste collegate**

Il secondo metodo per memorizzare i file è mantenerne ciascuno come una lista collegata di blocchi del disco (non necessariamente adiacenti), come appare nella Figura 4.11.

La prima parola di ciascun blocco è usata come puntatore al successivo. Il resto del blocco è costituito da dati.



Diversamente dall’allocazione contigua, con questo metodo può essere usato ogni blocco del disco. Non viene perso spazio in frammentazione del disco (eccetto che per la frammentazione interna nell’ultimo blocco). Inoltre è sufficiente che ciascuna voce delle directory memorizzi banalmente l’indirizzo del disco del primo blocco.

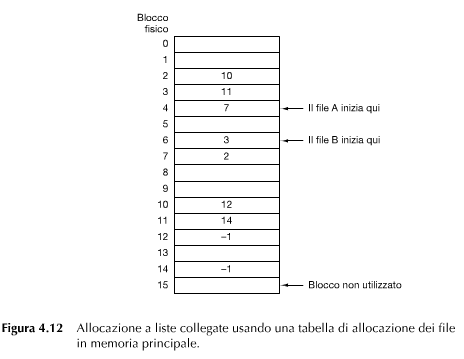
D’altra parte, sebbene leggere un file sequenzialmente sia semplice, l’accesso casuale è estremamente lento. Per avere il blocco *n*, il sistema operativo deve partire dal principio e leggere prima *n-1* blocchi, uno alla volta. Chiaramente fare tante operazioni di lettura in questo modo sarebbe incredibilmente lento.

Inoltre, la quantità di spazio per i dati di un blocco non è più una potenza di due, poiché il puntatore occupa alcuni byte. Dato che molti programmi leggono e scrivono in blocchi la cui dimensione è una potenza in base due, avere una dimensione peculiare risulta meno efficiente. Con i primi pochi byte di ciascun blocco occupati da un puntatore al blocco successivo, leggere l’intera dimensione del blocco richiede l’acquisizione e la concatenazione di informazioni da due blocchi del disco, il che genera ulteriore sovraccarico dovuto alla copia.

**Allocazione a liste collegate usando una tabella in memoria**

Entrambi gli svantaggi dell’allocazione a liste collegate possono essere eliminati prendendo la parola relativa al puntatore di ogni blocco del disco e mettendola in una tabella di memoria.

La Figura 4.12 mostra come si presenta la tabella relativa all’esempio della Figura 4.11.



In entrambe le figure abbiamo due file. Il file A usa i blocchi 4, 7, 2, 10 e 12, in quest’ordine e il file B usa i blocchi 6, 3, 11 e 14, in quest’ordine. Usando la tabella della Figura 4.12 possiamo partire con il blocco 4 e proseguire la sequenza sino alla fine. Lo stesso si può fare partendo dal blocco 6. Entrambe le sequenze sono terminate con un indicatore speciale (per esempio -1), che non è valido come numero di blocco. Una tabella siffatta in memoria principale è chiamata FAT (file allocation table).

Usando questa organizzazione, l’intero blocco è disponibile per i dati. L’accesso casuale è inoltre molto più semplice. Come nel precedente metodo, è sufficiente che la voce della directory gestisca un piccolo intero (il blocco di partenza) e sarà ancora in grado di localizzare tutti i blocchi, non importa quanto grande sia il file.

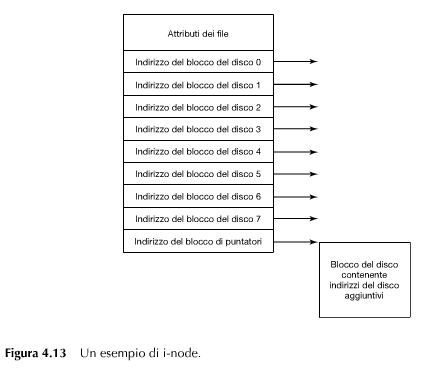
Lo svantaggio principale di questo metodo è che, affinché funzioni l’intera tabella, essa deve permanere in memoria. Con un disco da 1 TB e dimensioni blocco da 1 KB, la tabella necessita di un miliardo di voci, una per ciascuno del miliardo di blocchi del disco. Così facendo la tabella occuperà circa 3 GB di memoria.

L’idea della FAT non si presta bene a dischi di grosse dimensioni.

**I-node**

L’ultimo metodo che esaminiamo per tener traccia di quali blocchi appartengono a quale file è associato a una struttura dati chiamata i-node, che elenca gli attributi e gli indirizzi dei blocchi dei file.

Un semplice esempio è descritto nella Figura 4.13.



Dato un i-node, è possibile trovare tutti i blocchi di quel file. Il grande vantaggio di questo schema rispetto alle liste collegate che usano una tabella in memoria è che l’i-node ha bisogno di essere in memoria solo quando il file corrispondente è aperto. Se ciascun i-node occupa *n* byte e può essere aperto un massimo di *k* file contemporaneamente, la memoria occupata dall’array che tiene gli i-node è di soli *kn* byte per tutti i file aperti.

Questo array solitamente è molto più piccolo dello spazio occupato dalla tabella dei file descritta nel precedente paragrafo. La ragione è semplice: la tabella per tenere la lista collegata di tutti i blocchi disco è proporzionale alla dimensione del disco stesso (tante righe quanti i blocchi del disco); la dimensione dell’i-node invece richiede in memoria un array proporzionale al numero massimo di file che potrebbe essere contemporaneamente aperto.

Uno dei problemi degli i-node è che, se ogni i-node ha spazio per un numero fisso di indirizzi del disco, che cosa accadrà se un file supererà tale limite? Si riserva l’ultimo indirizzo del disco non per un blocco di dati, ma piuttosto per l’indirizzo di un blocco contenente ulteriori indirizzi di blocchi del disco, come mostrato nella Figura 4.13.

Ancora meglio sarebbe avere due o più di questi blocchi contenenti indirizzi del disco o anche blocchi del disco che puntano ad altri blocchi di disco pieni di indirizzi.

**Implementazione delle directory**

Al momento dell’apertura di un file, il sistema operativo usa il path name fornito dall’utente per localizzare la voce della directory. Questa fornisce le informazioni necessarie per trovare i blocchi del disco. A seconda del sistema questa informazione può essere l’indirizzo del primo blocco del disco (allocazione contigua e gli schemi di lista collegata) oppure il numero dell’i-node.

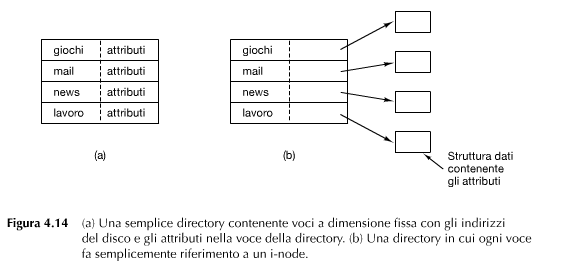
La directory associa il nome del file, gli attributi e il puntatore ai dati.

Una questione strettamente correlata è dove debbano essere memorizzati gli attributi.

Una possibilità è memorizzarli direttamente nella voce della directory. Questa opzione è illustrata nella Figura 4.14(a).

Un’altra possibilità, per i sistemi che li usano, è quella di memorizzare gli attributi negli i-node, invece che nelle voci di directory. Questo approccio è illustrato nella Figura 4.14(b).

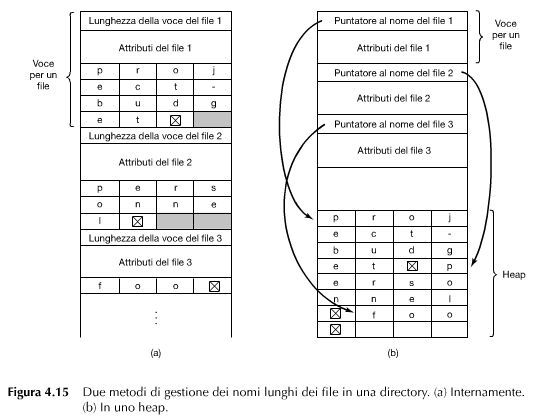
I due approcci mostrati nella Figura 4.14 corrispondono rispettivamente a Windows e UNIX.



Finora abbiamo supposto che i file abbiano nomi brevi e di lunghezza fissa. In MS-DOS i file hanno nomi da 1 a 8 caratteri e un’estensione opzionale di tre caratteri. In UNIX i nomi dei file hanno da 1 a 14 caratteri, inclusa qualunque estensione. Tuttavia quasi tutti i sistemi operativi moderni supportano nomi di file di lunghezza variabile. Com’è possibile implementarli?

L’approccio più semplice è di impostare un limite sul nome file, generalmente a 255 caratteri, e poi usare uno dei modelli della Figura 4.14 riservando 255 caratteri per ciascun nome di file. Questo sistema è semplice, ma spreca una buona parte dello spazio delle directory, poiché pochi file hanno nomi così lunghi. Per questioni di efficienza sarebbe opportuna una struttura diversa.

Un’alternativa è rinunciare all’idea che le voci di directory abbiano la stessa lunghezza. Con questo metodo, ciascuna directory contiene una parte di lunghezza fissa, che include gli attributi del file, e una parte di lunghezza variabile per il nome del file, come mostrato nella Figura 4.15(a).



Lo svantaggio di questo metodo è che quando il file viene cancellato, nella directory è introdotto un buco di ampiezza variabile, in cui il file successivo potrebbe non entrare. È lo stesso tipo di problema che abbiamo visto con i file contigui su disco, solo che adesso la compattazione della directory è fattibile, dato che sta tutta in memoria.

Un altro problema è che una singola voce di directory può estendersi su molteplici pagine, cosicché può accadere un page fault nel leggere il nome di un file.

Un altro modo per gestire nomi di lunghezza variabile è di fare tutte le voci di directory di lunghezza fissa e tenere i nomi dei file in uno heap alla fine della directory, come mostrato nella Figura 4.15(b). Questo metodo ha il vantaggio che quando è rimossa una voce, il successivo file inserito ci starà sempre. Naturalmente lo heap va gestito e possono comunque esserci page fault nel processare i nomi dei file.

Finora in tutti i modelli la ricerca nelle directory di un nome di un file avviene linearmente dall’inizio alla fine. Nelle directory estremamente estese la ricerca lineare può essere lenta.

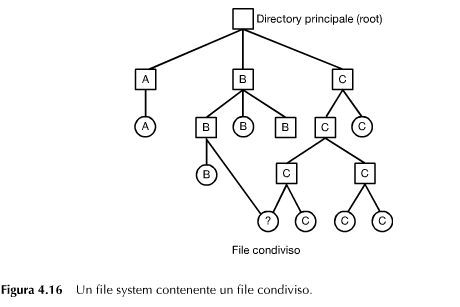
Un modo per velocizzarla è l’uso di una tabella di hash in ogni directory. Le tabelle hash hanno tempo di accesso velocissimi, ma lo svantaggio di una maggiore complessità nella gestione.

Un differente approccio per accelerare le ricerche è di utilizzare una memoria cache.

**File condivisi**

Quando parecchi utenti lavorano insieme a un progetto spesso hanno bisogno di condividere i file. Di conseguenza risulta comodo per un file condiviso apparire contemporaneamente in directory diverse, appartenenti a utenti diversi.

La Figura 4.16 mostra ancora il file system della Figura 4.7, solo con uno dei file di C ora presente anche in una delle directory di B. La connessione fra la directory di B e il file condiviso è chiamata link.



Il file system è ora un grafo aciclico orientato o DAG (directed acyclic graph), piuttosto che un albero.

La condivisione dei file è comoda, ma introduce anche dei problemi.

Innanzitutto, se le directory contengono realmente gli indirizzi del disco, quando il file viene collegato deve essere fatta una copia degli indirizzi nella directory di B.

Se B o C successivamente accodano qualcosa al file, i nuovi blocchi saranno elencati solo nella directory dell’utente che ha fatto l’append. I cambiamenti non saranno visibili all’altro utente, rendendo inutile così lo scopo della condivisione.

Questo problema è risolvibile in due modi. Nella prima soluzione i blocchi del disco non sono elencati nelle directory, ma in una piccola struttura dati associata al file (l’i-node). È l’approccio usato da UNIX. Nella seconda soluzione, il collegamento crea un nuovo file di tipo link che permette al sistema operativo di accedere al file sorgente, cioè quello di C. Questo approccio è denominato link simbolico.

Ciascuno di questi metodi ha i suoi svantaggi.

Nel primo metodo, quando B collega il file di C, l’i-node registra C come proprietario del file e il contatore dei riferimenti al file a 2 (Count). Se ora C rimuove il file, se si cancella l’i-node, B avrà una voce che punta ad un i-node non valido oppure, se l’i-node è stato riassegnato ad un altro archivio, ad un file errato. L’unica cosa possibile è cancellare la voce dalla directory di C e mantenere l’i-node in C (con Count = 1 e Owner = C). B è l’unico utente che ha una voce di directory per un file di proprietà di C.

Il secondo metodo presenta il problema dell’appesantimento della gestione (potrebbero essere necessari accessi al disco) ma possono essere utilizzate per collegare facilmente file che risiedono su macchine diverse.

**File system basati su log strutturati**

I cambiamenti nella tecnologia hanno spinto l’evoluzione dei file system: le CPU diventano sempre più veloci, i dischi sempre più grandi ed economici (ma non molto più veloci) e le memorie crescono esponenzialmente di dimensione. L’unico parametro a non migliorare considerevolmente è il tempo di ricerca dei dischi (per quelli meccanici).

Per questa ragione è stato progettato un nuovo tipo di file system: l’LFS (log-structured file system).

L’idea che guida la progettazione dell’LFS è che al velocizzarsi delle CPU e al crescere delle memorie, anche le cache dei dischi crescano rapidamente. Di conseguenza, la cache del file system può soddisfare una parte molto consistente di tutte le richieste di lettura direttamente senza bisogno di accedere al disco. Ne deriva che la maggior parte degli accessi al disco sarà in scrittura. Alcuni sistemi effettuano piccole scritture che sono altamente inefficienti: una scrittura da 50 µs impiega 10 ms per il tempo di seek e 5 ms per il ritardo di rotazione.

I progettisti dell’LFS decisero di reimplementare il file system UNIX a partire da queste considerazioni, in modo da sfruttare completamente l’ampiezza di banda disco, anche a fronte di un carico di lavoro composto in larga parte da piccole operazioni di scrittura casuali.

L’idea di base è di strutturare l’intero disco come un file di log. Periodicamente, e ogni volta che occorre, tutte le operazioni di scrittura da eseguire bufferizzate in memoria sono raccolte in un unico segmento e scritte sul disco come un singolo segmento contiguo alla fine del log. Un segmento singolo può contenere i-node, blocchi di directory e blocchi dati, tutti mischiati insieme. All’inizio di ciascun segmento si trova un segmento di sommario, che indica che cosa si trova nel segmento. Se si riesce a far stare il segmento medio nelle dimensioni di 1 MB, può essere sfruttata quasi tutta l’ampiezza di banda del disco.

Per facilitare la ricerca degli i-node, divenuto non più accessibile con l’i-number, si tiene una loro mappa, indicizzata per i-number. La voce *i* in questa mappa punta all’i-node *i* sul disco. La mappa è sia sul disco sia nella cache, così le parti più usate staranno per la maggior parte tempo nella memoria.

Se i dischi fossero infinitamente grandi, la descrizione appena fatta risulterebbe completa. Tuttavia i dischi reali hanno dei limiti di spazio, così il log potrebbe occupare l’intero disco, al punto da non poter scrivere nuovi segmenti nel log. Fortunatamente molti blocchi esistenti possono avere blocchi non più necessari; per esempio, se un file è sovrascritto, il suo i-node punterà adesso ai nuovi blocchi, ma i vecchi staranno ancora occupando spazio nei segmenti scritti in precedenza.

Per risolvere questo problema, l’LFS ha un thread cleaner (pulitore) che passa il suo tempo a fare una scansione circolare del log per compattarlo.

Le misure fornite nella documentazione sull’argomento mostrano che l’LFS supera UNIX di un

ordine di grandezza sulle piccole operazioni di scrittura, mentre ha prestazioni uguali o migliori di UNIX sulle operazioni di lettura e sulle operazioni di scrittura estese.

**File system journaling**

Sebbene i file system basati sui log strutturati siano un’idea interessante, il loro impiego non è molto diffuso a causa della loro incompatibilità con i file system già esistenti.

L’idea di base è quella di tenere un log di ciò che il file system sta per fare prima che lo effettui in modo che, in caso di crash prima del lavoro programmato, il sistema stesso sia in grado di verificare, una volta riavviato, ciò che stava facendo al momento del crash e terminare il lavoro.

Questo tipo di file system, detto file system journaling, è ampiamente utilizzato. I file system NTFS di Microsoft, ext3 di Linux c ReiserFS utilizzano il journaling.

Affinché il journaling funzioni occorre che le operazioni nel log devono essere idempotenti: possono essere ripetute tutte le volte che serve senza produrre effetti collaterali.

Operazioni del tipo “aggiorna la bitmap per contrassegnare l’i-node *k* o il blocco *n* come liberi” possono essere ripetute finché tutto sia andato a buon fine senza problemi. Analogamente, la ricerca di una directory e la rimozione di una voce denominata *foobar* è allo stesso modo idempotente.

Per migliorare l’affidabilità, un file system può introdurre il concetto di transazione atomica: più operazioni possono essere raggruppate in una entità unica e svolta in modo atomico.

**File system virtuali**

I sistemi operativi sono in grado di gestire diversi file system.

Un sistema Windows potrebbe avere un file system principale NTFS, ma anche un drive FAT-32 o FAT-16. Windows gestisce questi file system identificando ciascuno con una differente lettera di unità (come C:, D:, …). Quando un processo apre un file, la lettera dell’unità è esplicitamente o implicitamente presente, così Windows sa a quale file system passare la richiesta. Non c’è alcun tentativo di integrare file system eterogenei in uno solo unificato.

Al contrario, tutti i sistemi UNIX moderni fanno un tentativo molto serio di integrazione di molteplici file system in una struttura singola. Un sistema Linux potrebbe avere ext2 come il file system di root, con una partizione ext3 montata su /usr e un secondo disco rigido con un file system RaiserFS montato su /home e così via..

La maggior parte dei sistemi UNIX hanno usato il concetto di un VFS (virtual file system) per cercare di integrare più file system in una struttura ordinata. L’idea chiave è di astrarre quella parte di file system comune a tutti i file system e mettere quel codice in un layer separato che richiama i reali file system sottostanti per gestire i dati.

Il VFS ha un’interfaccia “inferiore” verso i file system reali, denominata interfaccia VFS.