Gestione della memoria secondaria

Index

Introduction

Il SO è responsabile dell'assegnamento di blocchi a file, ma ci sono due problemi correlati (influenzati l'un l'altro):

- occorre allocare spazio per i file, e mantenerne traccia una volta allocato

- occorre tenere traccia dello spazio allocabile

I file si allocano in “porzioni” o “blocchi” (non byte a byte per motivi di efficienza) la cui dimensione minima è il settore del disco, ma di solito ogni porzione o blocco è una sequenza contigua di settori

---

Allocazione di spazio per i file

Per l’allocazione di spazio per i file ci sono vari problemi da affontare:

- decidere se fare **preallocazione** o **allocazione dinamica**
- decidere se lo spazio deve essere diviso in parti di dimensione fissa (**blocco**) o di dimensione dinamica (**porzione**)
- quale deve essere il metodo di allocazione: **contiguo**, **concatenato** o **indicizzato**
- gestione della *file allocation table* (per ogni file, serve a mantenere le informazioni su dove si trovano le porzioni che lo compongono sul disco)

> ⚠ **Warning**
>
> In Unix non esiste una File Allocation Table vera e propria, si utilizzano gli inode

---

Preallocazione vs. allocazione dinamica

Quando si parla di **preallocazione** si intende che la dimensione massima di un file viene dichiarata a tempo di creazione, che risulta facilmente stimabile in alcune applicazioni (es. risultato di compilazioni, file che forniscono sommari sui dati) ma difficile su molte altre infatti utenti ed applicazioni tendono a sovrastimare la dimensione risultando in uno spreco di spazio su disco, a fronte di un modesto risparmo di computazione.

Per questo motivo viene quasi sempre preferita l’**allocazione dinamica**, tramite la quale la dimensione del file viene aggiutasta in base alle call `append` (aggiungere informazioni) o `truncate` (rimuovere informazioni)

Dimensioni delle porzioni

Per decidere la dimensione delle porzioni si hanno due possibilità agli estremi:

- si alloca una **porzione larga a sufficienza per l'intero file** → efficiente per il processo che vuole creare il file (viene allocata della memoria contigua)
- si alloca **un blocco alla volta** → efficiente per il SO, che deve gestire tanti file, ma ciascun blocco è una sequenza di n settori contigui con n fisso e piccolo (spesso $n = 1$)

Si cerca dunque un punto di incontro (*trade-off*) tra efficienza del singolo file ed efficienza del sistema.
Sarebbe ottimo, per le prestazioni di accesso al file, fare porzioni contigue, ma inefficiente per l'SO; invece porzioni piccole vuol dire grandi tabelle di allocazione, e quindi grande overhead, ma vuol dire anche maggior facilità nel riuso dei blocchi.
Sarebbe inoltre da evitare di fare porzioni fisse grandi in quanto porterebbe a frammentazione interna, ma rimane possibile la frammentazione esterna in quanto i file possono essere cancellati

Alla fine risultano quindi rimanere due possibilità (valide sia per preallocazione che per allocazione dinamica):

- **porzioni grandi e di dimensione variabile**
  - ogni singola allocazione è contigua
  - tabella di allocazione abbastanza contenuta
  - complicata la gestione dello spazio libero: servono algoritmi ad hoc
- **porzioni fisse e piccole**
  - tipicamente, 1 blocco per una porzione
  - molto meno contiguo del precedente
  - spazio libero: basta guardare una tabella di bit

Con la preallocazione viene naturale utilizzare porzioni grandi e di dimensione variabile. Infatti con questa combinazione non è necessaria la tabella di allocazione dato che per ogni file basta l'inizio e la lunghezza (ogni file è un'unica porzione) e come per il partizionamento della RAM si parla di best fit, first fit, next fit (ma qui non c'è un vincitore troppe variabili ed è molto inefficiente per lo spazio libero in quanto necessita periodica compattazione che è molto più onerosa rispetto a quella per la RAM).

Come allocare spazio per i file

Per allocare spazio per i file si utilizzano tre metodi:

- **contiguo**
- **concatenato**
- **indicizzato**

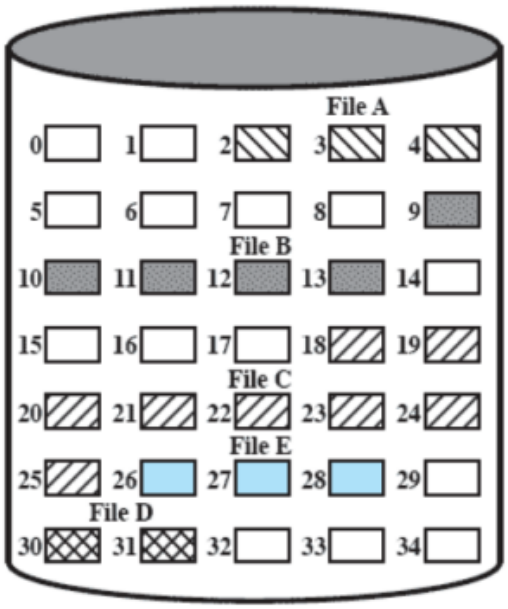
Per ciascuno di questi tre metodi ci sono delle caratteristiche

	Contiguous	Chained	Indexed	
Preallocation?	Necessary	Possible	Possible	
Fixed or variable size portions?	Variable	Fixed blocks	Fixed blocks	Variable
Portion size	Large	Small	Small	Medium
Allocation frequency	Once	Low to high	High	Low
Time to allocate	Medium	Long	Short	Medium
File allocation table size	One entry	One entry	Large	Medium

Allocazione contigua

Con l'allocazione contigua un insieme di blocchi viene allocato per il file quando quest'ultimo viene creato. Dunque la preallocazione risulta necessaria, occorre infatti sapere quando lungo, al massimo sarà il file, altrimenti, se un file può crescere oltre il limite massimo potrebbe incontrare blocchi già occupati

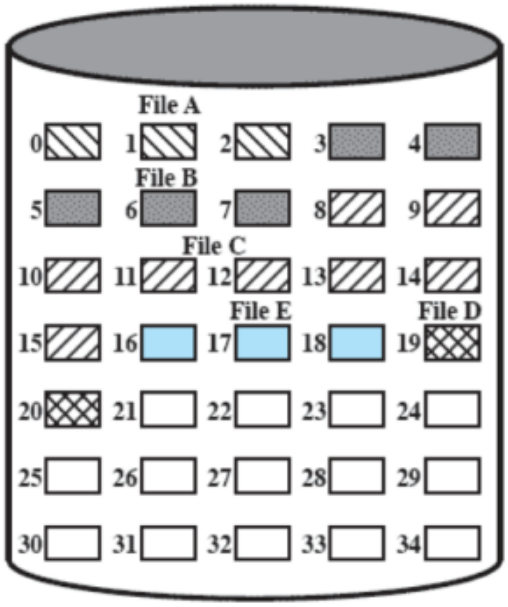
In questo modo risulta necessaria una sola entry nella tabella di allocazione dei file (blocco di partenza e lunghezza del file).
Tipicamente ci sta frammentazione esterna, con conseguente necessità di compattazione



File Allocation Table

File Name	Start Block	Length
File A	2	3
File B	9	5
File C	18	8
File D	30	2
File E	26	3

Compattazione



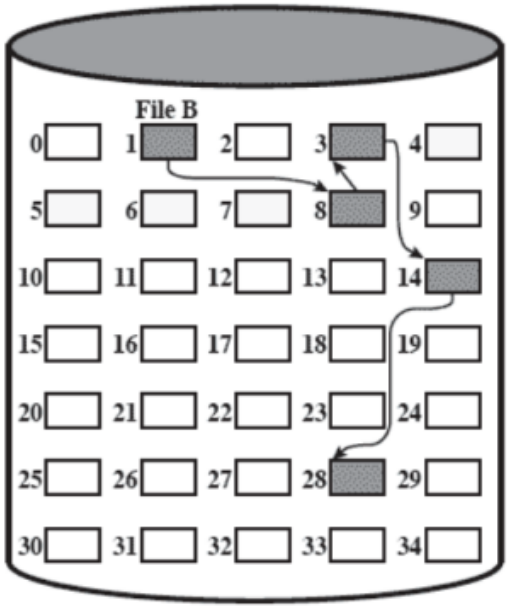
File Allocation Table

File Name	Start Block	Length
File A	0	3
File B	3	5
File C	8	8
File D	19	2
File E	16	3

Allocazione concatenata

Con l'allocazione concatenata viene allocato un blocco alla volta che ha un **puntatore al prossimo blocco** (la prima parte del blocco sono dati del file, l'ultima, piccola, parte del blocco è il puntatore)

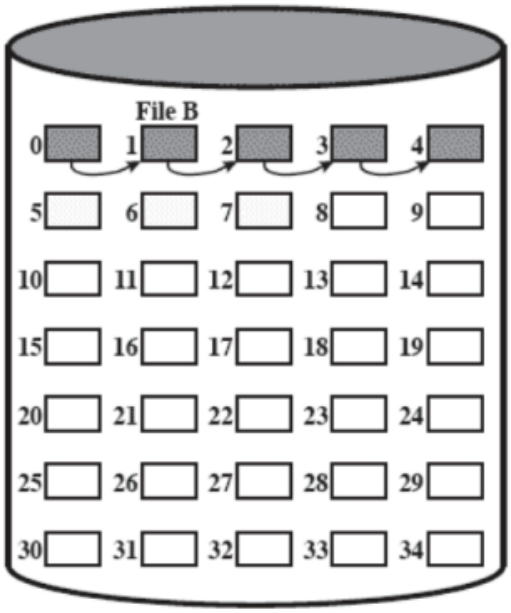
Risulta quindi necessaria **una sola entry** nella tabella di allocazione dei file contenente il blocco di partenza e la lunghezza del file (lunghezza anche calcolabile, ma è comodo avere già il valore calcolato)
In questo modo **non si ha frammentazione esterna** (la frammentazione interna è trascurabile, in quanto molto piccola)
Con questo tipo di allocazione non risulta essere un problema accedere ad un file in modo sequenziale, ma se serve un blocco che si trova b blocchi dopo quello iniziale, occorre scorrere tutta la lista. Per risolvere questo problema si ricorre al **consolidamento**, analogo alla compattazione, per mettere i blocchi di un file contigui e migliorare l'accesso sequenziale



File Allocation Table

File Name	Start Block	Length
• • •	• • •	• • •
File B	1	5
• • •	• • •	• • •

Consolidamento



File Allocation Table

File Name	Start Block	Length
• • •	• • •	• • •
File B	0	5
• • •	• • •	• • •

Allocazione indicizzata

L'allocazione indicizzata è quella che, in seguito ad alcune modifiche, viene utilizzata nei computer odierni. Risulta essere una **via di mezzo** tra l'allocazione contigua e quella concatenata, risolvendo quasi tutti i problemi che le precedenti avevano.

La tabella di allocazione dei file contiene, apparentemente **una sola entry**, con l'indirizzo di un blocco. Questo blocco, in realtà, ha **una entry per ogni porzione allocata al file** (quindi fa parte della tabella a tutti gli effetti pur trovandosi in un blocco apparentemente indistinguibile da quelli usati per i dati del file).

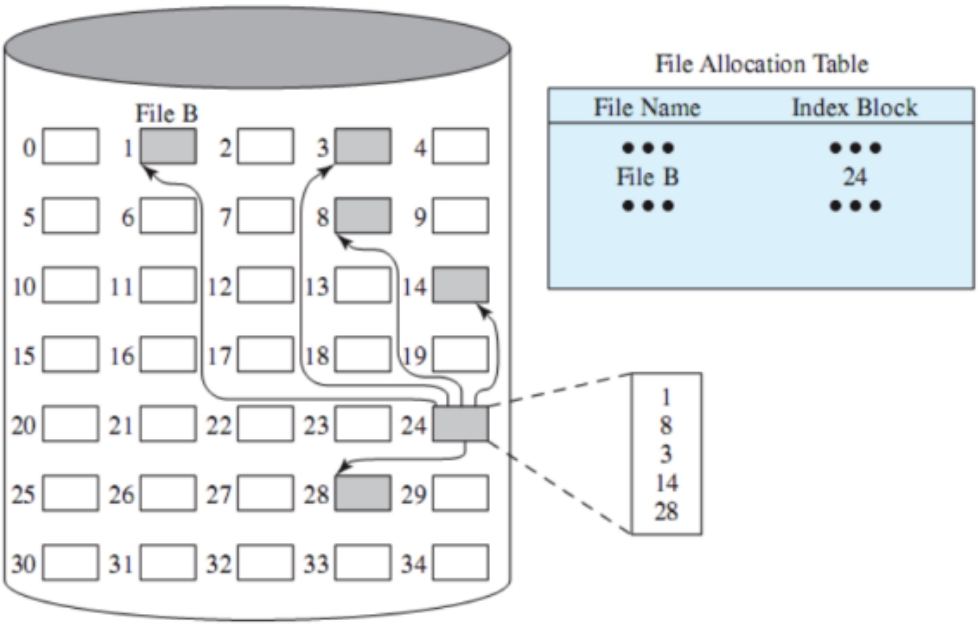
E se un file è troppo grande e quindi in un solo blocco non si riescono a contenere tutti gli indici agli altri blocchi contenenti i dati del file? Si fanno più livelli (es. inode di Unix-Linux).

Ovviamente ci dev'essere un **bit** che dica se un **blocco è composto da dati o è un indice**

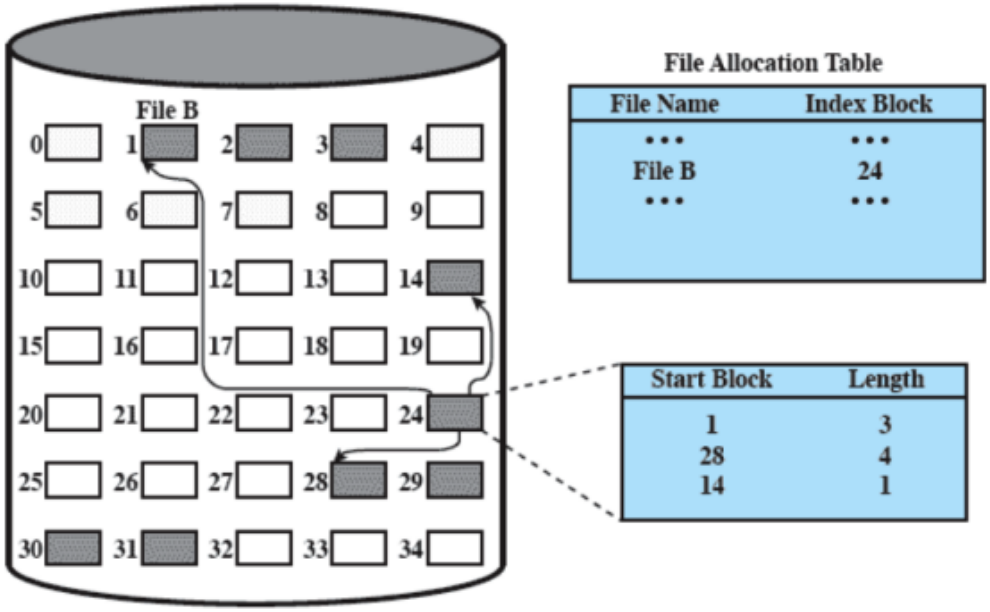
L'allocazione può essere con:

- blocchi di **lunghezza fissa** → niente frammentazione esterna
- blocchi di **lunghezza variabile** → migliora la località
  A volte occorre il consolidamento:
- blocchi di lunghezza fissa → migliora la località
- blocchi di lunghezza variabile → riduce la dimensione dell'indice

Porzioni di lunghezza fissa



Porzioni di lunghezza variabile



Gestione dello spazio libero

Come è importante gestire lo spazio libero, è altrettanto importante gestire anche quello occupato

Per allocare spazio per i file, ci sono diverse tecniche, quella più naturale e semplice da pensare sarebbe quello di fare il complemento dei blocchi presenti all'interno della allocation table, ma non è realistico. Per questo motivo serve una struttura dati dedicata, che tenga traccia dell'**allocazione di disco** (oltre a quella per i file)

> **Hint**
>
> Mentre per l'allocazione del file, quello viene caricato in RAM solo una volta aperto, è necessario che si sappia in RAM (almeno in minima parte) quanto e quale sia lo spazio libero sul disco

Dunque ogni volta che si alloca o cancella un file, la tabella di allocazione di disco va aggiornata

Abbiamo dunque vari metodi:

- **tabelle di bit**
- **porzioni libere concatenate**
- **indicizzazione**
- **lista dei blocchi liberi**

Tabelle di bit

Viene creato un vettore con **un bit per ogni blocco su disco**, con `0` se libero e `1` se occupato

Viene in questo modo minimizzato lo spazio richiesto e risulta compatibile con ogni schema visto finora, ma se il disco è quasi pieno, la ricerca di uno spazio libero può richiedere molto tempo (anche se questo problema è in parte risolvibile con delle tabelle riassuntive, mantengo infatti in RAM delle tabelle riassuntive in cui ho uno zero se tot blocchi sono tutti liberi, mentre un uno nel caso in cui almeno uno di questi è occupato, riducendo di molto lo spazio occupato)

Porzioni libere concatenate

Le **porzioni libere**, usando un ragionamento complementare all'allocazione concatenata, possono essere **concatenate le une alle altre** usando, per ogni blocco

libero, un puntatore ed un intero per la dimensione (numero di blocchi liberi successivi a quello considerato). In questo modo viene sostanzialmente eliminato l'overhead di spazio

Ci sono però alcuni problemi, infatti: se ci sta molta frammentazione allora le porzioni sono tutte quante da un blocco e la lista diventa lunga ed è molto lungo cancellare file molto frammentati (infatti non si può rimuovere una sola entry di questa tabella, ma bisogna eliminarne molteplici)

Indicizzazione

Allo stesso modo, usando un ragionamento complementare all'indicizzazione dello spazio occupato, si può **indicizzare lo spazio libero** (i file-system principali di Linux utilizzano questo metodo)

Lista dei blocchi liberi

In questo caso, a differenza delle porzioni libere concatenate, c'è una qualche zona di memoria che devo riservare a parte in cui si mette una **lista di indici corrispondenti ai blocchi liberi**.

Nonostante possa sembrare poco efficiente in realtà l'overhead a livello di spazio non è grande; supponiamo che per ogni indirizzo siano necessari $4\ \text{bytes}$ e i blocchi siano da $512\ \text{bytes}$, vuol dire che richiederebbe l'1% di spazio su disco.
Però con questo tipo di allocazione, si riesce abbastanza bene a fare in modo di avere parti di queste liste in memoria principale.

Dunque questa questa lista può essere organizzata come pila; per allocare spazio libero bisogna fare un `pop` (quello che prima era libero ora non lo è più), quando invece devo deallocare dello spazio allocato devo fare un `push` e guardo la cima per vedere gli spazi liberi. Se si fanno troppi `pop` (ho esaurito lo spazio libero nella porzione caricata in RAM) mi basta caricare la prossima porzione di lista in RAM

Volumi

Un volume è essenzialmente quello che viene chiamato un disco *"logico"*, ovvero una **partizione di un disco** che può essere trattata in **maniera indipendente dal file system** (oppure più dischi messi insieme e visti come un disco solo, LVM)

Dunque i volumi sono un insieme di settori in memoria secondaria che possono essere usati dal SO o dalle applicazioni

I settori di un volume non devono necessariamente essere contigui, ma appariranno come tali al SO e alle applicazioni
Un volume potrebbe essere il risultato dell'unione di volumi più piccoli

---

Dati e metadati

Dati → contenuto di un file
Metadati → lista blocchi liberi, lista blocchi all'interno del file, data di ultima modifica, proprietario, …

I metadati, come i dati, si devono trovare su disco, perché devono essere persistenti, ma per efficienza vengono anche tenuti in memoria principale.
Però mantenere **consistenti i metadati** in memoria principale e su disco è **inefficiente**, quindi si fa solo di tanto in tanto, quando il disco è poco usato e con più aggiornamenti insieme.
Questa tecnica è chiamata *journaling* e consiste di anziché di fare le modifiche a dati e metadati non appena si aggiornano, **si tiene traccia di cosa è stato modificato** in una zona di disco dedicata (*log*), per poi scrivere tutto insieme in un secondo momento (in caso di reboot dopo un crash, basta leggere il log)

Se il computer viene spento all'improvviso senza una procedura di chiusura (es. mancanza di corrente) o se il disco viene rimosso senza dare un appropriato comando (`unmount`) potrebbe succedere che il journaling mi possa dare problemi.
Per risolvere questo tipo di problemi mi basta scrivere un bit all'inizio del disco, che dice se il sistema è stato spendo correttamente; al reboot, se il bit è 0, occorre eseguire un programma di ripristino del disco, operazione assai complessa senza il journaling (basta il log!!)

Journaling

Per il journaling è necessaria zona dedicata del disco in cui scrivere le operazioni, prima di farne il commit nel file system. Generalmente questo viene implementato come log circolare in modo tale da evitare di avere file system corrotti infatti: se la scrittura nel journal è completa, in caso di crash durante la scrittura nel file system è possibilerecuperare l'errore, se c'è un crash durante la scrittura nel journal, il file system rimane integro

Recupero dati

Se al reboot il bit di shutdown è `0`:

1. Confronta il journal allo stato corrente del file system
2. Correggi inconsistenze nel file system basandosi nelle operazioni salvate nel journal
    - se la scrittura nel journal è completa, in caso di crash durante la scrittura nel file system è possibile recuperare l'errore
    - se c'è un crash durante la scrittura nel journal, il file system rimane integro

Tipi di journaling:

- **fisico** → copia nel journal tutti i blocchi che dovranno poi essere scritti nel file system, inclusi metadati; se c'è un crash durante la scrittura nel file system, è sufficiente copiare il contenuto dal journal al file system al reboot successivo
- **logico** → copia nel journal soltanto i metadati delle operazioni effettuate (es. modifica blocchi liberi dopo cancellazione file); se c'è un crash, si copiano i metadati dal journal al file system, ma questo può causare corruzione dei dati (append a file modifica la lunghezza, ma il contenuto è perso perché non salvato nel journal)

Alternative

- Soft Updates File Systems → Riordina scritture su file system in modo da non avere mai inconsistenze, o meglio, consentono solo alcuni tipi di inconsistenze che non portano a perdita di dati (*storage leaks*)
- Log-Structured File Systems → L'intero file system è strutturato come un buffer circolare, detto log; dati e metadati sono scritti in modo sequenziale, sempre alla fine del log e ci possono essere diverse versioni dello stesso file, corrispondenti a diversi istanti temporali
- Copy-on-Write File Systems → Evitano sovrascritture dei contenuti dei file; scrivono nuovi contenuti in blocchi vuoti, poi aggiornano i metadati per puntare ad i nuovi contenuti