giovedì 22 giugno 2023

ARCHITETTURE DI CALCOLO LEZIONE 24

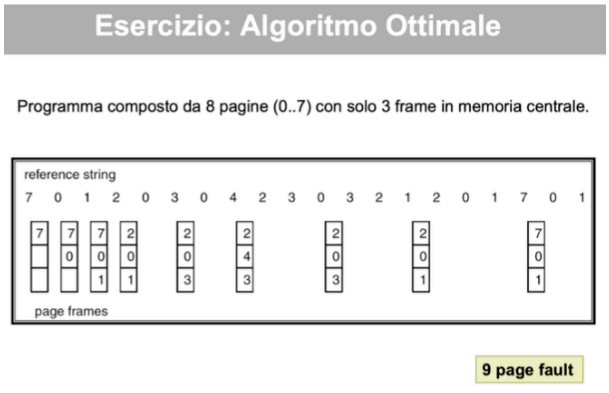
Esercizi sulla sostituzione delle pagine, implementazione del file system

Esercizi sulla sostituzione delle pagine

1. Esercizio con algoritmo FIFO
Date le seguenti pagine di un programma (sequenza 7, 0, 1 ,2, 0, ...,1), bisogna inserire in modo appropriato tutte le pagine nei tre frame a disposizione, sostituendo secondo l'algoritmo FIFO. Tale algoritmo prevede che la pagina che ha occupato un frame per primo sia la prima ad essere sostituita.

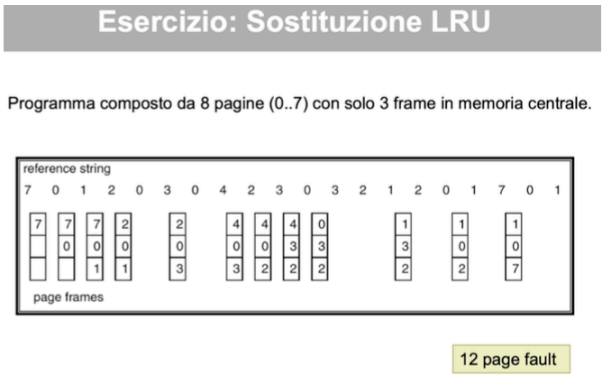
Si parte dal riempimento dei tre frame, ottenendo la sequenza 7 0 1; quando arriva il 2 bisogna sostituire la pagina che ha occupato un frame per prima, in questo caso la pagina 7: si ottiene la sequenza 2 0 1. La pagina 0 è già presente nella sequenza, quindi, non c'è bisogno di eseguire alcuna sostituzione (è già allocata in memoria). Quando arriva la pagina 3, questa va a sostituirsi alla pagina 0 e così via, fino al termine della sequenza. Si vanno a contare i page fault, ossia i casi in cui una pagina non allocata in memoria va a sostituirsi con una già allocata. In questo esempio abbiamo 15 page fault (RICORDA, anche i caricamenti iniziali contano come page fault!).
Trucchetto: deve essere sostituita la pagina che viene ripetuta n volte, dove n è il numero di frame disponibile.

2. Esercizio con algoritmo ottimale



L'algoritmo ottimale ha come criterio la sostituzione della pagina che, guardano la sequenza in avanti, verrà usata "più in là" nel tempo. Per esempio, dopo aver caricato 7 0 1, la pagina 2 andrà a sostituire la pagina 7, in quanto la 0 e la 1 verranno caricate prima cronologicamente parlando. Allo stesso modo, la pagina 3 sostituirà la pagina 1, poi la pagina 4 sostituirà la pagina 0 e così via, fino alla fine della sequenza. I page fault verificatisi sono 9.

3. Esercizio con algoritmo LRU



L'algoritmo LRU prevede la sostituzione della pagina che è stata caricata nel frame meno di recente, secondo la logica che le pagine chiamate più di recente possono essere "vicine" delle pagine che verranno chiamate dopo. Dopo il caricamento della sequenza 7 0 1, la pagina 2 va a sostituirsi alla pagina7; segue la sostituzione della pagina 1 con la 3, poi della 2 con la 4 e così via. Alla fine, si ottengono complessivamente 12 page fault.

Soluzione esercizi dati a lezione

1. Si consideri la seguente successione di riferimenti a pagine in memoria centrale:
   1, 2, 3, 2, 1, 4, 5, 6, 6, 4, 2, 1, 2, 3, 4

Calcolare quante assenze di pagine (page fault) si verificano se si usano 3 blocchi di memoria con i seguenti algoritmi di sostituzione:

- LRU
- Ottimale

LRU

1	2	3	2	1	4	5	6	6	4	2	1	2	3	4
1	1	1			1	1	6			6	1		1	4
	2	2			2	5	5			2	2		2	2
		3			4	4	4			4	4		3	3

Numero di page fault: 10

Ottimale

1	2	3	2	1	4	5	6	6	4	2	1	2	3	4
1	1	1			1	5	6				1		1	
	2	2			2	2	2				2		3	
		3			4	4	4				4		4	

Numero di page fault: 8

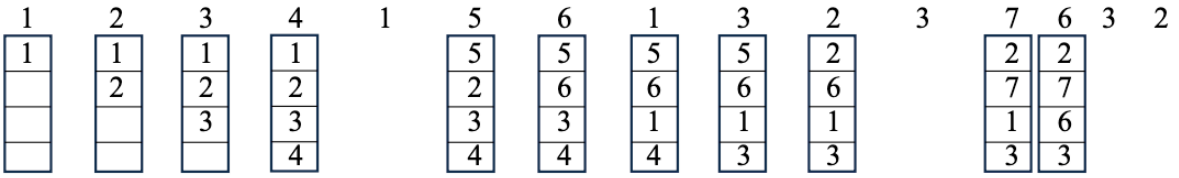
- (nell'ultima sostituzione si è dovuto decidere se far prendere al 3 il posto dell'1 o del 2; non avendo informazioni su futuri inserimenti in memoria, si applica l'algoritmo FIFO)

2. Si consideri la seguente successione di riferimenti a pagine in memoria centrale:
1, 2, 3, 4, 1, 5, 6, 1, 3, 2, 3, 7, 6, 3, 2

Calcolare quanti page fault si verificano se si usano 4 blocchi di memoria con i seguenti algoritmi di sostituzione:

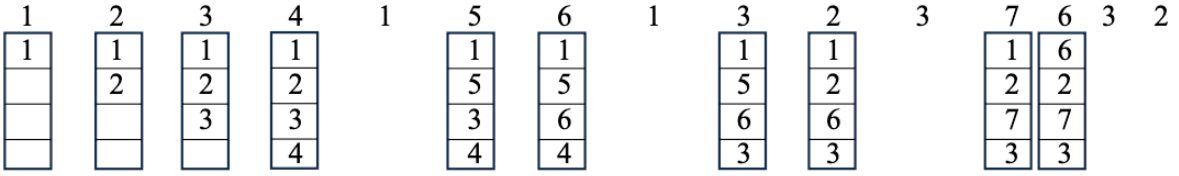
• FIFO
• LRU

FIFO



Numero di page fault: 11

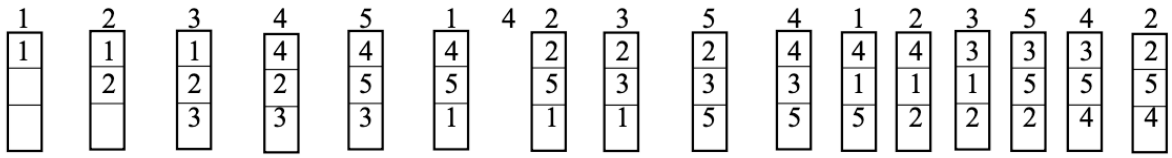
LRU



Numero di page fault: 10

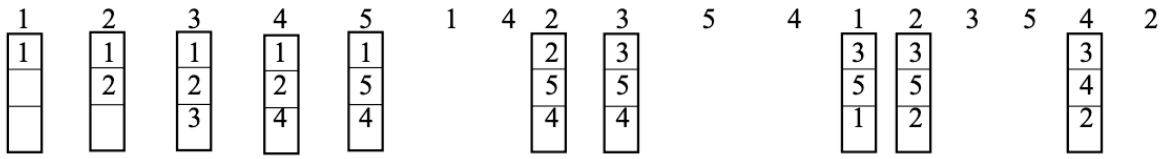
3. Data una memoria centrale gestita mediante paginazione, si consideri la seguente sequenza di richieste di pagine:
1, 2, 3, 4, 5, 1, 4, 2, 3, 5, 4, 1, 2, 3, 5, 4, 2
Calcolare il numero di page fault che si verificano applicando gli algoritmi FIFO e ottimale usando tre frame in memoria centrale.

FIFO



Numero di page fault: 16

Ottimale



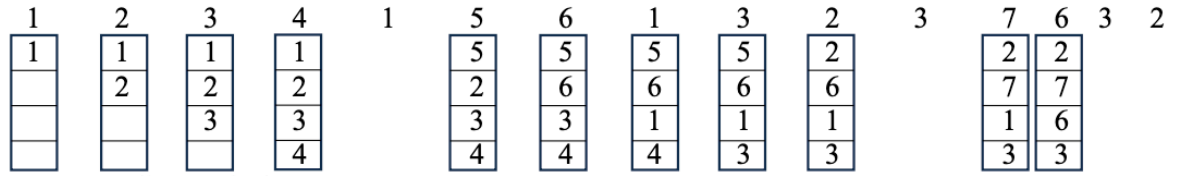
Numero di page fault: 10

(il 4 prende il posto del 5 per l'algoritmo FIFO)

4. Si consideri la seguente successione di riferimenti a pagine in memoria centrale:
1, 2, 3, 4, 1, 5, 6, 1, 3, 2, 3, 7, 6, 3, 2

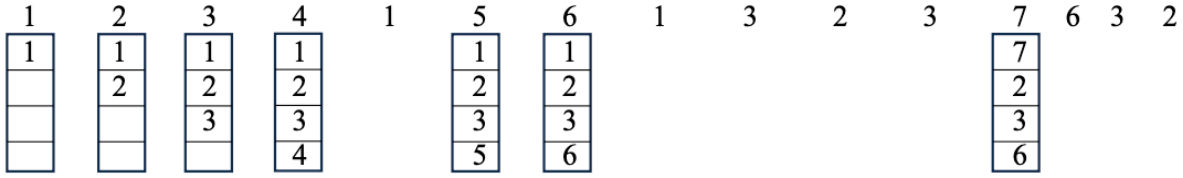
Calcolare quante assenze di pagine (page fault) si verificano se si usano 4 blocchi di memoria con i seguenti algoritmi di sostituzione:
• FIFO
• Ottimale

FIFO



Numero di page fault: 11

Ottimale



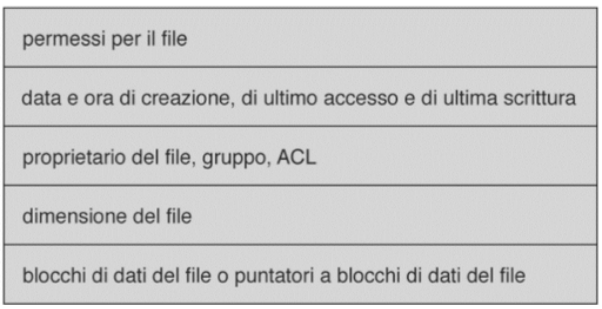
Numero di page fault: 7

FILE SYSTEM

Per realizzare un file system si usano parecchie strutture dati, sia nei dischi sia in memoria. Queste strutture variano a seconda del sistema operativo e del file system. Fra le strutture presenti nei dischi ci sono:

- Blocco di controllo di avviamento: contiene le informazioni posizionali dei file necessari per l'avviamento del S.O. Qualora vi sia un danno di tale sistema, il S.O. non si avvia correttamente e la macchina non funziona.
- Blocchi di controllo dei volumi: contengono dettagli riguardanti la partizione, quali numero totale dei blocchi e loro dimensione, contatori dei blocchi liberi e relativi puntatori, contatore degli FCB (File Control Block) liberi e relativi puntatori. L'FCB è il corrispettivo dei file del process control block (informazioni sullo sheduling, registri ecc.); esso contiene informazioni relativi alla gestione di un file.
- Strutture delle directory: usate per organizzare i file.

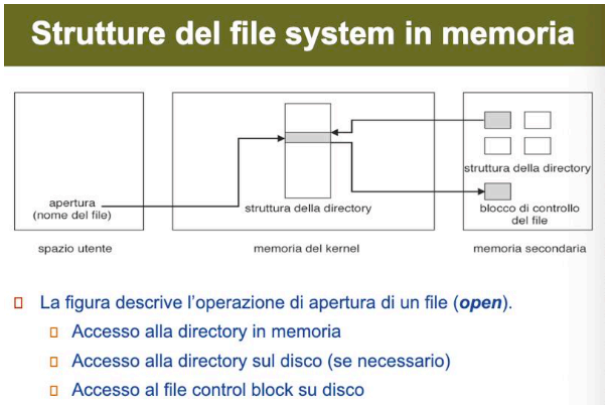
File Control Block (FCBC)



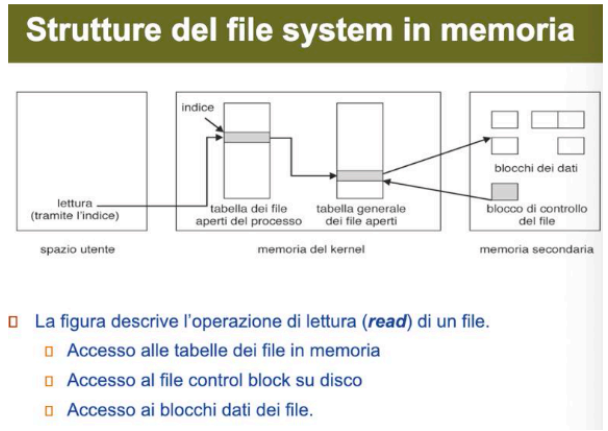
Contiene le informazioni riguardanti un file. In UNIX prende il nome di inode (index node), poiché può essere considerato come un puntatore ai blocchi di un disco che, di fatto, formano un grafo (il file system). Nell'immagine troviamo i dati contenuti nell'FBC di un file.
Tali dati sono persistenti, cioè si trovano su disco. Le informazioni che, invece, risiedono nella RAM sono:

- Tabella di montaggio: contiene le informazioni relative a ciascun volume (Disco C, Disco D, pennetta USB, ecc.) montato in un dato momento.

- Struttura delle directory: contiene informazioni relative a tutte le directory a cui i processi hanno avuto accesso di recente (cache). Queste informazioni, "cloni" RAM di ciò che è contenuto in disco, aiutano a velocizzare i tempi di risposta.
- Tabella dei file aperti: copia dell'FCB dei file aperti nel sistema.
- Tabella dei file aperti per ciascun processo



L'immagine a sinistra mostra come le strutture della RAM e del disco sono implicate nel processo di apertura di un file. Supponiamo che nello spazio di un programma apriamo un certo file tramite la system call "open". Tale chiamata va a guardare nella struttura della directory nella RAM (memoria kernel); da qui, si trova il percorso del file e se necessario si va sul disco. Se l'informazione necessaria all'apertura del file richiesto è presente nella cache allora, tramite puntatori, si va nei blocchi richiesti.
Aprire un file della directory la prima volta comporta più tempo delle volte successive in quanto, dopo la prima apertura, il file viene salvato nella cache, accelerando il tempo di apertura.



Anche nel caso di un'operazione di "read" sono richieste strutture dati della RAM e strutture dati del disco. L'operazione di read permette di leggere un file in seguito ad una "open".
Innanzitutto, si accede alla tabella dei file in memoria tramite l'handle, ossia un indice che identifica il processo; si passa, tramite l'indice, alla tabella generale dei file aperti e, da qui, ai blocchi dei dati e FCB.

File system virtuali

Per rendere le directory indipendenti dal tipo di S.O. utilizzato, sono stati ideati i cosiddetti virtual file system (VFS). La VFS è un'astrazione del file system, ideata nei sistemi preesistenti UNIX, che permette di usare una stessa interfaccia (API) per differenti tipi di file system. Alcuni sistemi operativi, infatti, permettono all'apertura di scegliere il tipo di file system che l'utente vuole usare (per esempio in base al tipo di file con i quali si vuole lavorare).

L’interfaccia opera verso il VFS che “nasconde” i diversi tipi di file system sottostanti. La struttura del file system prevede:
- Interfaccia del file system: linea di comando della gestione dei file
- Interfaccia del VFS: traduce lerichieste “dall’alto” in comandi specifici; può tradurre anche comandi da un tipo S.O. ad un altro (Esempio: macchina LINUX legge file Windows).
- Diversi tipi di file system
- Memoria

Inoltre, tramite la VFS un file system remoto può essere aggiunto ai file system locali, permettendo
di accedere ai file di tale macchina al momento remota ed accedervi come se fossero file locali.

Implementazione della directory

La scelta del metodo di allocazione e gestione delle directory ha un impatto sull'efficienza e sull'affidabilità del file system. Due sono i metodi principali:
- **Lista lineare** dei nomi dei file con i puntatori ai blocchi dei dati (contenuto dei file)
    - semplice da programmare
    - non molto efficiente nella ricerca (lineare)
    - Lista ordinata (B-tree): migliora il tempo di ricerca, ma l'ordinamento deve essere mantenuto a fronte di ogni inserimento/cancellazione
- **Tabella hash** — lista lineare con struttura hash per la ricerca.
    - Migliora il tempo di ricerca nella directory
    - Inserimento e cancellazione costano O (1), se non si verificano collisioni. Collisione: situazione in cui due nomi di file generano lo stesso indirizzo hash nella tabella
    - Dimensione fissa e necessità di rehash (o liste di trabocco); il rehash è una funzione che raddoppia lo spazio di memoria allocato per la tabella

Un metodo di allocazione si occupa di come allocare sulla memoria secondaria i blocchi di un file. Esistono tre metodi principali:
- **Allocazione contigua**: ciascun file occupa un insieme di blocchi contigui sul disco
- **Allocazione concatenata**: ciascun file è costituito da blocchi, anche distanti tra loro e collegati grazie ai puntatori
- **Allocazione indicizzata**: i blocchi indice danno informazioni sulla posizione dei “veri” blocchi di ciascun file

Allocazione contigua

Nell’allocazione contigua, per reperire un file, occorrono solo la locazione iniziale (# blocco iniziale) e la lunghezza (numero di blocchi). È possibile effettuare sia l’accesso sequenziale che l’accesso casuale (si conosce la lunghezza L dei blocchi quindi, per andare al blocco n, si fa L x n) Questo algoritmo ha, però, delle problematiche:

- Frammentazione esterna (dovendo allocare i blocchi gli uni vicini agli altri, è possibile che vi sia uno spazio totale sufficiente a contenerli ma che questo non sia necessariamente contiguo)
- Si possono impiegare gli stessi algoritmi di allocazione della RAM (per il best-fit è necessario molto tempo)
- Compattazione dello spazio disco (o deframmentazione): richiede una grande quantità di tempo

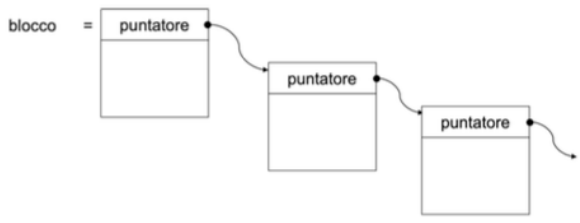
Come avviene la traduzione da indirizzo logico a indirizzo fisso di un file?
Se si vuole operare su un file, bisognerà individuare il blocco in cui è stato inserito e, per fare ciò, è necessario conoscere la grandezza del blocco e il punto d'inizio del file.
Supponiamo che un singolo blocco abbia dimensione di 512 byte e che il file di interesse sia il 2500; sarà necessario, allora, eseguire la seguente divisione: 2500:512, di cui il quoziente rivela il numero del blocco (blocco 4) mentre il resto indica la posizione del file all'interno del blocco stesso (posizione 883). È possibile utilizzare questo metodo grazie alla contiguità dei blocchi.

Allocazione contigua modificata

Quando si crea un file si sa che questo potrebbe crescere di dimensioni nel tempo, come, ad esempio, una tesina. Questa situazione viene gestita dall'allocazione contigua modificata. In questo caso nei sistemi operativi di nuova generazione sono state inventate delle estensioni dei blocchi. Tuttavia, questo meccanismo delle extension non può essere considerato particolarmente efficiente soprattutto per file molto grandi; inoltre, introduce il problema della frammentazione interna quando l'estensione che si va a creare è di dimensioni maggiori rispetto alla porzione restante di file da allocare.

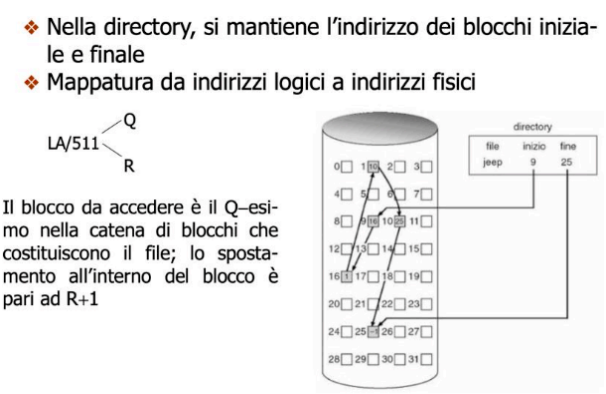
Allocazione concatenata

Si preferisce, dunque, utilizzare l'allocazione concatenata. In questo caso, ciascun file è una lista concatenata di blocchi: i blocchi possono essere sparsi ovunque nel disco e ogni blocco contiene un puntatore che indirizza al blocco successivo, tranne l'ultimo che, invece, conterrà un simbolo che indica che quello è l'ultimo blocco.
Il vantaggio dell'allocazione concatenata è che non spreca spazio, in quanto non esiste la frammentazione esterna.



Maggiore è la dimensione del blocco, maggiore è lo spreco che si ha sull'ultimo blocco. Ad esempio, se si decide di utilizzare un blocco da 4096 byte, invece di uno da 512 byte, e si ha un file da 1 byte, questo occupa ugualmente 4096; quindi 4095 byte risulteranno sprecati.
Un problema dell'allocazione concatenata è che non supporta l'accesso diretto.

Se si vuole andare al blocco numero 40, bisogna passare attraverso tutti i blocchi precedenti tramite il puntatore.
Che cosa memorizziamo?



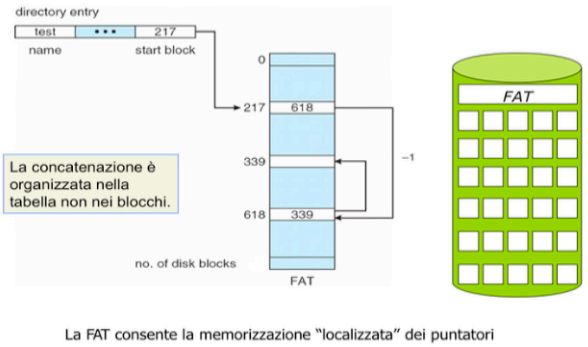
L'indirizzo del blocco iniziale e finale. Nell'immagine il blocco 9 è quello iniziale, il 25 è il blocco finale. Si memorizza il blocco finale poiche ci possiamo aggiungere cose alla fine senza passare per tutti i blocchi precedenti. Come si fa a capire in che blocco devo andare? La formula è prendere l'indirizzo logico, dividere per 511(perché 1 byte è occupato dal puntatore) si otterrà un quoziente (numero del blocco a cui si arriva sequenzialmente) e un resto (posizione interna).

File Allocation Table (FAT)

È una variante del metodo di allocazione concatenata, implementata in MS−DOS e OS/2.
FAT si basa su principi molto semplici:

- Per contenere la FAT si riserva una sezione del disco all'inizio di ciascun volume
- La FAT ha un elemento per ogni blocco del disco ed è indicizzata dal numero di blocco
- L'elemento di directory contiene il numero del primo blocco del file
    - L'elemento della FAT indicizzato da quel blocco contiene a sua volta il numero del blocco successivo del file
    - L'ultimo blocco ha come elemento della tabella un valore speciale di fine file.
- I blocchi inutilizzati sono contrassegnati dal valore 0.
- Facilita l'accesso casuale: informazione relativa alla locazione di ogni blocco è "concentrata" nella FAT

Descrizione dello schema:

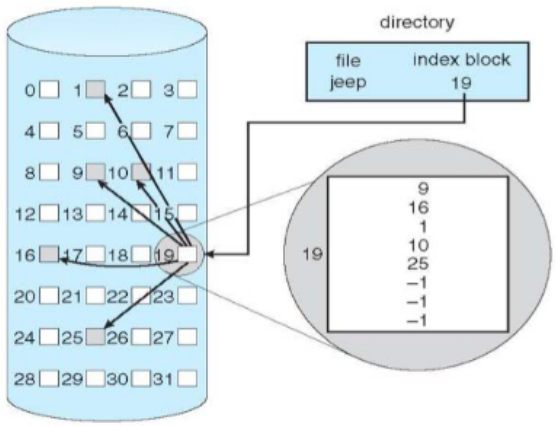


La FAT consente la memorizzazione "localizzata" dei puntatori

Si supponga di avere una directory con un file denominato "test"; implementando la FAT, di questo file ci è sufficiente conoscere solo qual è il blocco iniziale. Se, per esempio il blocco iniziale è il 217, allora, andando nella FAT mi sarà indicato il blocco successivo, ovvero il 618, e quello ancora dopo (il 339), e così via.
Questo sistema permette di far muovere solo minimamente la testina del disco in quanto, sebbene debba attraversare tutti gi indici dei blocchi precedenti a quello di interesse nella FAT, questo movimento è nettamente inferiore rispetto alla scansione che sarebbe altrimenti necessaria dei blocchi in memoria.

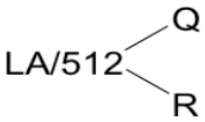
Allocazione indicizzata

È implementata nei sistemi UNIX. Raggruppa tutti i puntatori ai blocchi dei file in un'unica struttura: il blocco indice (index table), memorizzata nella directory.
Questo sistema di allocazione richiede una tabella indice che indichi dove sono situati i blocchi di dati che costituiscono il file. Permette l'accesso dinamico senza frammentazione esterna; tuttavia, si può verificare un sovraccarico temporale di accesso al blocco indice.
Esempio:



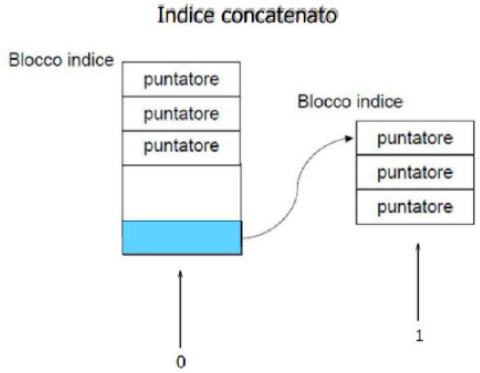
Il blocco indice 19 riporta la posizione dei blocchi nell'ordine con cui si susseguono; in questo caso il primo sarà il 9, a seguire il 16, poi l'1, e così via.
Il blocco indice occupa spazio; infatti, se, per esempio, si ha un file system con blocchi da 4096 byte l'uno, quanto occupa un file su disco? 4096 x 2, perché un blocco è occupato dal blocco indice mentre l'altro blocco dai dati.
I vantaggi dell'allocazione indicizzata sono:
•Supporta l'accesso diretto
•Non c'è frammentazione esterna.
•Il mapping è molto semplice

Mapping da indirizzo logico a indirizzo fisico nel caso in cui il blocco indice sia contenuto in un solo blocco:



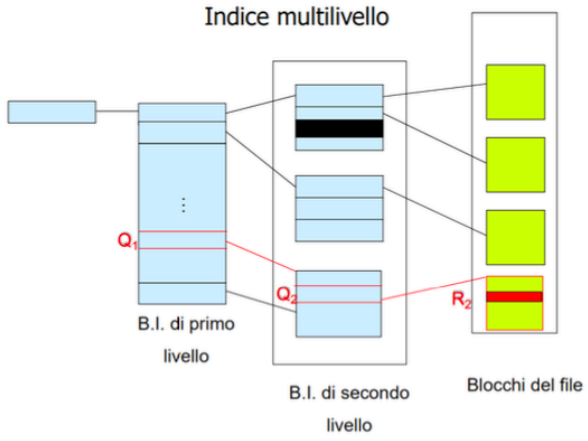
Quoziente (Q)= numero del blocco.
Resto (R)= posizione del file all'interno del blocco

Se un blocco non è sufficiente a contenere il blocco indice di un file, per i blocchi indice si possono utilizzare:
- schema concatenato
- schema multilivello
- schema combinato



Una versione alternativa dell'allocazione indicizzata si basa sul concetto di indice concatenato.
Se si ha un blocco indice che è saturo, è sufficiente aggiungere nell'ultima posizione un puntatore che condurrà a un altro blocco indice e che, a sua volta, potrà portare ad un altro blocco indice (in parole semplici, l'ultima posizione del blocco porta a un altro blocco)

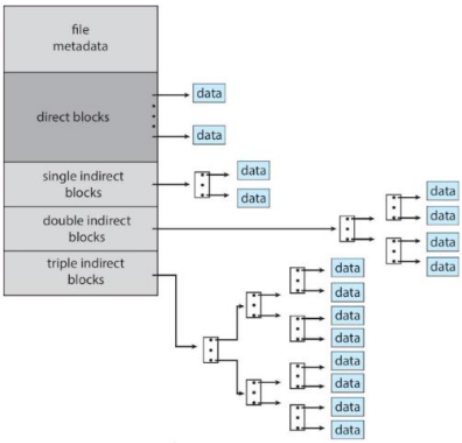
Indice multilivello



Nella directory è presente un puntatore che porta a blocchi indice di primo livello, che, a loro volta, portano a blocchi indice di secondo livello, per finire al blocco dei dati vero e proprio.
I livelli sono infiniti, ma non possiamo esagerare, in quanto la tipologia è esponenziale e per questo motivo non si va oltre l'indicizzazione a 2 livelli.

Inode Unix

Cerca di sfruttare il meglio di tutte le tecniche viste fin ora. Dall'immagine si vede un inode che rappresenta un file unix.



All'inizio ci sono metadata classici, poi i primi 12 di questi 15 puntatori puntano a blocchi diretti, cioè, contengono direttamente gli indirizzi di blocchi contenenti dati del file. Quindi, i dati di piccoli file (non più di 12 blocchi) non richiedono un blocco indice distinto. Se la dimensione dei blocchi è di 4 KB, è possibile accedere direttamente fino a 48 KB di dati.
Gli altri tre puntatori puntano a blocchi indiretti. Il primo è un puntatore a un blocco indiretto singolo; si tratta di un blocco indice che non contiene dati, ma indirizzi di blocchi che contengono dati.

Il secondo è un puntatore a un blocco indiretto doppio contenente l'indirizzo di un blocco che a sua volta contiene gli indirizzi di blocchi contenenti puntatori agli effettivi blocchi di dati.
In fine a triplo livello.
Perché utilizzare questo approccio?
Se il file è piccolo, e quindi rientra in soli 12 blocchi, non c'è bisogno di andare a usare il blocco indice per memorizzare i blocchi successivi, Molti sono i file piccoli all'interno del sistema mentre pochi i file di grandi dimentisioni, quindi la maggior parte sono tutti gestiti direttamente nei primi 12 blocchi, mentre man mano che diventano di maggior dimensione si entra nei blocchi a 1 livello, a 2 livelli o per i file di dimensioni estreme si utilizza il blocco a 3 livelli.
Qual è il miglior metodo di allocazione?

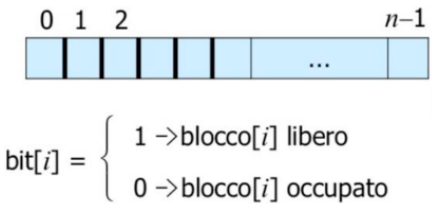
La prima scelta è capire se si preferisce un accesso sequenzionale o diretto; nel caso dell'accesso diretto bisogna utlizzare una tecnica contigua o concatenata, mentre, per la frammentazione esterna bisogna utilizzare un approccio indicizzato (attualmente il più utilizzato) oppure la versione dell'Inode Unix.
Il metodo indicizzato, anche se più complesso, è molto più vantaggioso rispetto ai pochi svantaggi che comporta, come l'utilizzo leggermente maggiore di spazio.

Deframmentazione

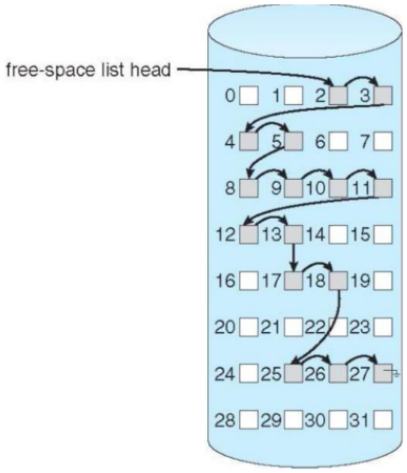
La deframmentazione è un'operazione di ottimazione che consente di ridurre la frammentazione esterna dei file presenti sulla memoria stessa. Soprattutto sui dischi magnetici, la deframmentazione permette di ridurre il tempo impiegato dal controller del disco per posizionare la testina sulla traccia in cui risiede un determinato blocco; quindi, non vi sarà più il bisogno di attendere che il piatto del fisco ruoti finché il blocco non raggiunge la testina. Questo problema non è presente nei dischi SSD, basati su memoria flash ad accesso casuale e dunque non subisce ritardo. La memoria flash può essere scritta solo per un numero limitato di volte e dunque la deframmentazione potrebbe essere dannosa.

Gestione dello spazio libero

Il sistema operativo per trovare lo spazio necessario alla memorizzazione di un file utilizza o vettori di bit o liste concatenate.



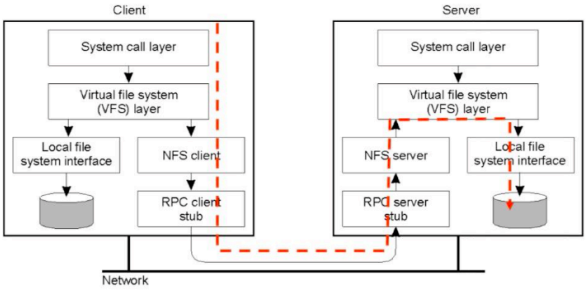
Un vettore di bit o bitmap è un array contenente bit (0/1) la cui posizione i-esima vale 1 se è libera e 0 se è occupato. Quindi quando arriva un file di dimensione n blocchi, si calcola dal numero del primo blocco libero scorrendo il vettore, cercando il primo byte/parola diverso/a da 0. Si hanno buone prestazioni se il vettore è conservato in memoria centrale.



Il secondo approccio è quello di tenere la lista concatenata dei blocchi liberi, ovvero si collegano tutti i liberi mediante puntatori e si mantiene un puntatore alla testa della lista in memoria centrale. Con questo approccio non si spreca spazio e non è necessario attraversare tutta la lista, poiché di solito la richiesta è relativa ad un singolo blocco.
File system distribuiti
Un file system distribuito è un file system che permette di accedere a file presenti su macchine diverse. Ne esistono numerosi (NFS, AFS, Coda, Plan9, xFS)

Network file system (nfs)

Originariamente sviluppato dalla Sun Microsystems per le sue workstation con sistema operativo UNIX, si trova facilmente anche su Linux. Nasce dall'idea di integrare file system differenti, ovvero di accedere a file remoti di file system diversi.



Se un computer vuole accedere ad un altro computer per accedere ad un file, diventa un server NFS che permette ad un client di accedere al file e rimandarlo indietro. Il client non si accorge che sta lavorando su un file remoto, in quanto utilizza gli stessi comandi locali sul file stesso (scaricamento, modifica e caricamento file) in maniera del tutto trasparente e visibile all'utente finale.

I due computer sono collegati da una rete e sia client che server hanno un layer (uno strato) VFS di accesso al file system. A seconda del file a cui si vuole accedere si traduce la richiesta:

- al file system locale, se si vuole il file locale;
- se da remoto passa la richiesta al VFS layer, un modulo locale in grado di colloquiare con un modulo remoto. In particolar modo il VFS layer utilizza una remote procedure call (chiamata di procedura remota) che chiama il server di ascolto sulla rete, arriva all'NFS server passa dal VFS layer e prende il file locale del sistema remoto.

Dal punto di vista del client, la richiesta è più veloce ad essere servita rispetto dall'essere fatta in remoto, ma dal punto di vista dei comandi e della qualità delle risposte non vi è alcuna differenza.