File System

Ogni applicazione deve poter salvare informazioni e riprenderle in futuro, ogni informazione deve essere salvata dopo la terminazione del processo. Come possiamo salvare, trovare e mantenere queste informazioni?

- Il SO crea una astrazione attraverso il **file system**.

Dal punto di vista di un utente un file è l'unica comunicazione che si ha con i settori di un hard disk. Non possiamo scrivere sui settori ma possiamo scrivere i file.

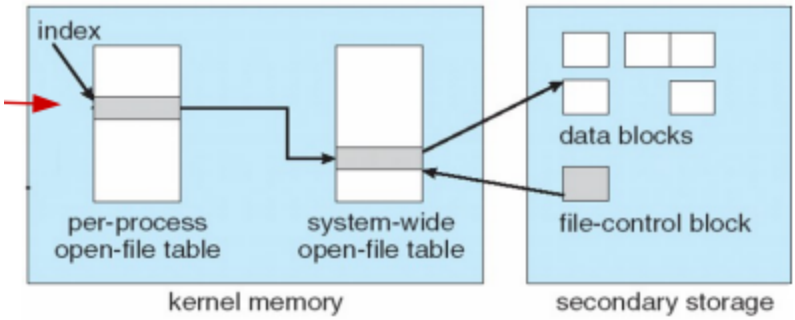
Le informazioni sono salvate in maniera persistente.

Il file system si interfaccia con le periferiche attraverso i driver, per salvare, trovare e interagire con i file.

Layout del file system

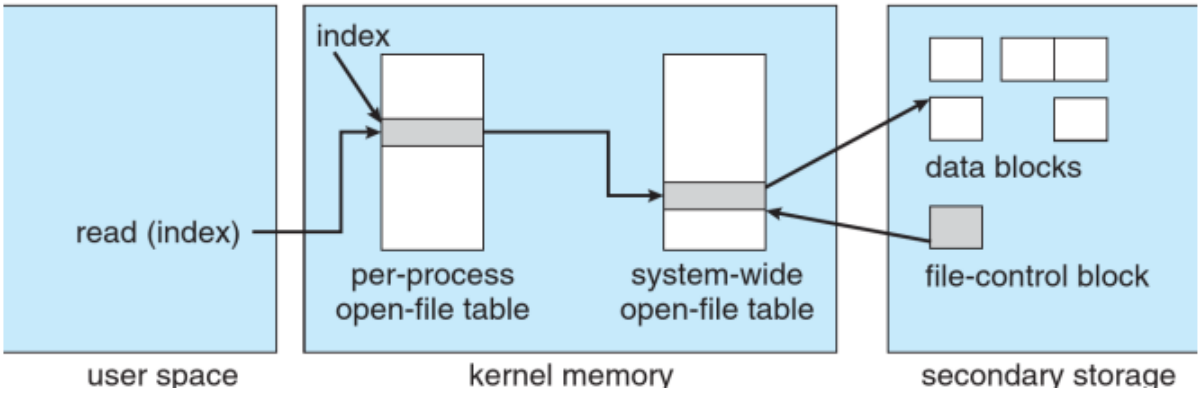
La prima operazione è la **create** di un file. Per implementare la funzione di create crea un nuovo file control block lo aggiungo alla directory entry e punta al file che ho creato.

La seconda operazione è la **open**, bisogna verificare che il file esista e quindi si vede se il file c'è e dopo si prende il suo file control block. Il sistema operativo si salva il puntatore al file.



Secondary storage è la memoria di massa, quando viene fatta una open il sistema operativo aggiunge una riga al file che stiamo aprendo, viene aggiunta una riga alla tabella dei file aperti, viene copiato il suo file control block. La tabella dei file aperti punta ai dati di quel file.

Quando viene fatta la **close** vengono distrutti i puntatori, quindi rimossa la riga viene chiaramente distrutto anche il puntatore.



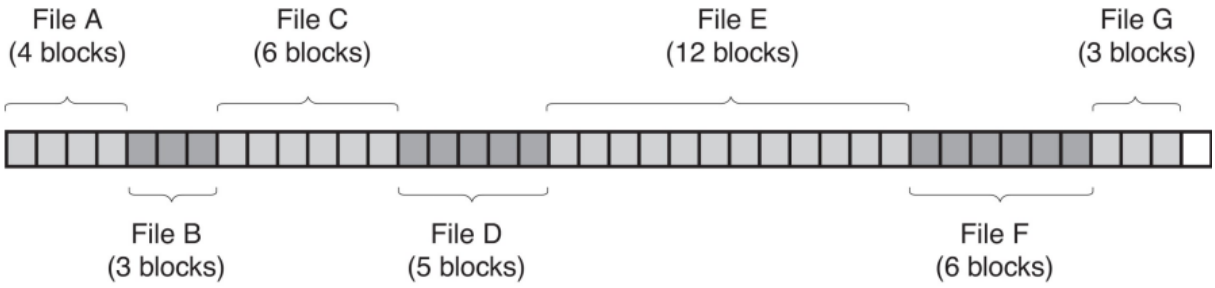
Ogni processo ha una sua tabella che punta alla tabella dei file aperti, così facendo possono leggere e scrivere lo stesso file, non assieme chiaramente perchè c'è poi il problema dei lettori e scrittori.

Un file è una sequenza di blocchi nella data area nel file system, come possiamo allocarli al meglio? Grazie agli schemi di allocazione dei file.

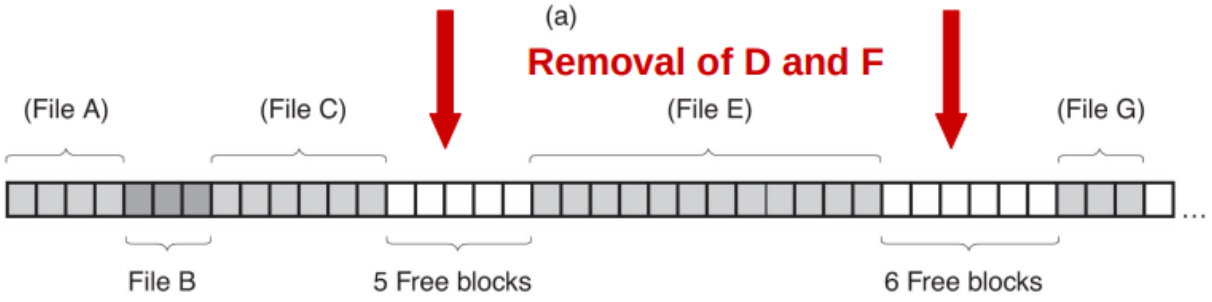
Schema 1: Allocazione contigua

Vuole minimizzare le seek dei dischi meccanici, il modo migliore è allocare i blocchi su settori consecutivi nella stessa traccia al peggio nello stesso cilindro. Memorizziamo un file come una sequenza:

- File da 50Kib:
  - Blocchi dei dischi da 1Kib → gli dobbiamo dare 50 blocchi consecutivi
  - Blocchi dei dischi da 2Kib → gli dobbiamo dare 25 blocchi consecutivi
- Serve quindi l'indirizzo del primo blocco e il numero di blocchi nel file.



Ha molti punti a favore che vengono coperti dal un grande problema ovvero della frammentazione esterna, quando rimuoviamo un file creiamo dei buchi:



Abbiamo due soluzioni:

- **Deframmentazione**: compattiamo i dischi spostandoli per non lasciare buchi, così facendo lo spazio che prima si era creato dall'eliminazione ora è di nuovo tutto unito.

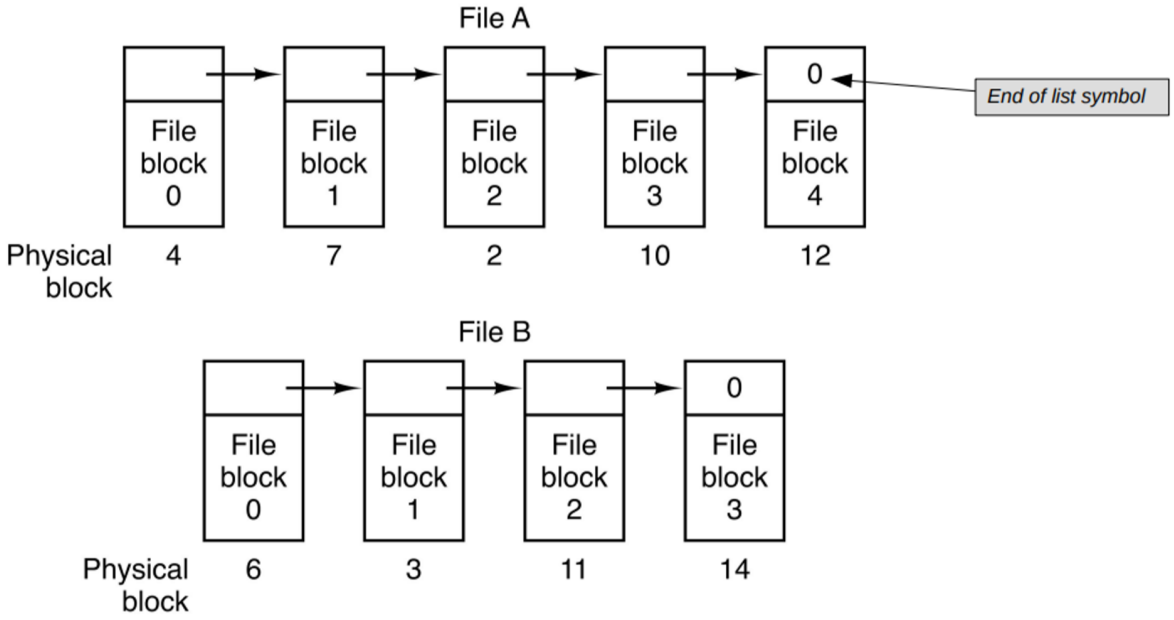
Schema 2: Allocazione Extent

Si ragiona con regioni dette extent quando un file è creato uno spazio contiguo di blocchi liberi è allocato, se la sua dimensione cresce uno spazio è aggiunto alla regione (solitamente il doppio della regione). Le regioni sono di dimensione variabile. Attenzione che la regione è composta di zone non contigue ma a livello logico lo sono.

Per tenere traccia di tutti gli extent e di come sono composti ci vuole tempo e spazio, in più ci sono problemi di frammentazione perchè se sono di dimensioni fisse ci sarà una frammentazione interna, se invece è lunghezza variabile non ha una minima lunghezza (≥ 2) ci sarà una frammentazione esterna.

Schema 3: Linked list

I blocchi sono organizzati come una linked list quindi appena si salva il primo blocco possiamo viaggiare su tutta la lista di blocchi che formano il file.



Non c'è frammentazione esterna e il file può crescere la sua lunghezza fino all'intera partizione, l'accesso sequenziale può richiedere molte seek se non sono nella stessa traccia. Stessa cosa non possiamo avere accessi casuali perchè dobbiamo sempre leggere la testa. Questo schema è pessimo.

C'è anche una discrepanza tra lettura e dimensioni effettive del blocco dato che abbiamo uno spazio piccolo nel blocco per il puntatore al blocco successivo. Così facendo dovremmo leggere due blocchi invece che uno solo.

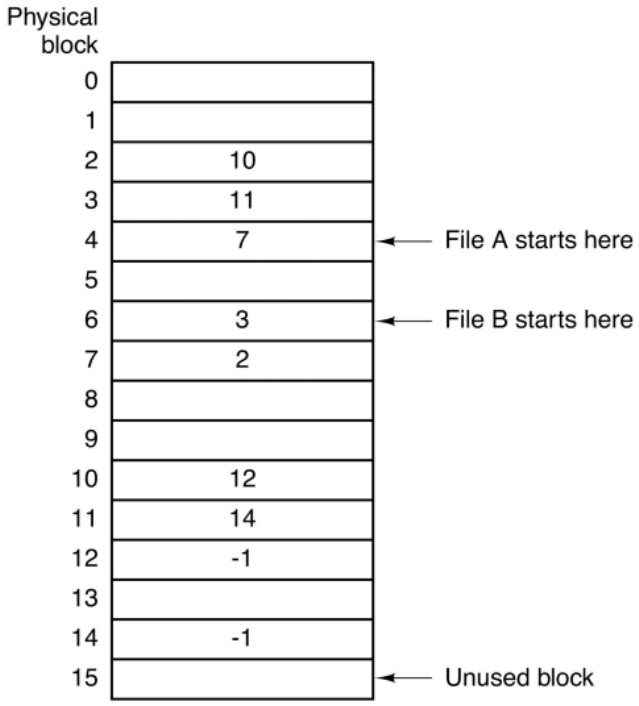
Se perdiamo un blocco l'intera catena va a farsi fottere, volevo dire, diventa inutilizzabile.

Schema 4: Allocazione Cluster

Questa è una estensione della linked list, ora abbiamo in un nodo più blocchi di memoria, chiamati cluster, così facendo abbiamo delle performance migliori rispetto alla linked list ma ha un enorme problema di frammentazione esterna. Quindi abbiamo settori riuniti in blocchi e ora abbiamo blocchi riuniti in cluster, se dobbiamo assegnare solo un byte perdiamo un sacco di spazio. Questa soluzione è stata ideata da un idiota per fare un paper universitario, inutile.

Schema 5: Linked List con FAT

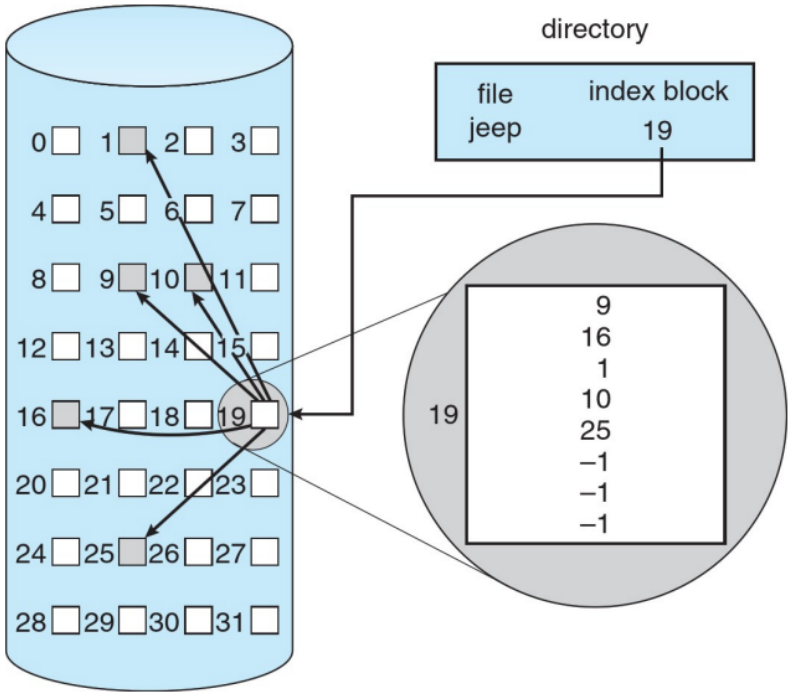
Mette i puntatori in una tabella (**File Allocation Table**), quindi ogni riga ha un puntatore al prossimo blocco oppure un simbolo speciale per la fine del file o che è un blocco libero.



Il problema di fare le seek è che ci si impiega il doppio del tempo, per risolverlo si butta la FAT in memoria centrale → occupa troppo spazio!

Schema 6: Allocazione con indirizzo

Ogni file ha una sua tabella di allocazione (chiamata index block) salvata in uno o più blocchi del disco. Quando facciamo accesso alla directory entry di quel file avremo il blocco in cui è salvata la index block, così facendo avremo un blocco che punta ad altri blocchi (una mini FAT).



Esercizio

- Block pointer: 4B
- Disk block: 1KiB
- Index length: 1 disk block, 4 disk blocks
- *Max file size?*

- With 1KiB index block:
  → $\lfloor 2^{10}B / 2^{2}B \rfloor = 2^{8}$ entries
  → Max file size: 2^{8} entries × 1KiB = 256KiB

- With 4KiB index block:
  → $\lfloor 2^{12}B / 2^{2}B \rfloor = 2^{10}$ entries
  → Max file size: 2^{10} entries × 1KiB = 1MiB