05a - File System

File

Introduzione

Il file system fornisce i meccanismi sia per l'accesso ai dati e ai programmi residenti nei dischi, sia per la loro registrazione.

Un file è una raccolta d'informazioni tra loro correlate definite dal suo creatore. È un tipo di dato astratto, con un insieme di valori ed un insieme di operazioni effettuabili su di essi; quindi definire un file significa definire la sua struttura, i suoi attributi e le operazioni alle quali può essere soggetto.

Dal punto di vista dell'utente, un file è la più piccola porzione di memoria secondaria logica; i dati si possono cioè scrivere in memoria secondaria soltanto all'interno di un file.
Fornisce agli utenti (e in particolare ai programmatori) una interfaccia uniforme per la gestione dei dispositivi di I/O.

Nella gestione delle memorie di massa, ogni file indica uno spazio logico degli indirizzi contiguo, indipendentemente da come è realizzato.

Il sistema operativo gestisce la corrispondenza tra i file e i dispositivi che li contengono fisicamente; i file sono normalmente organizzati in directory che ne facilitano l'uso.

Struttura dei file

Un file ha una struttura definita secondo il tipo (un file testo, sorgente, oggetto, eseguibile ecc). I tipi di file si possono anche utilizzare per indicare la struttura interna dei file.
I file sorgente e i file oggetto hanno una struttura corrispondente a ciò che il

programma che dovrà leggerli si attende; inoltre alcuni file devono rispettare una determinata struttura comprensibile al SO. Alcuni SO impongono (e gestiscono) un numero minimo di strutture di file. Questo orientamento è stato seguito da UNIX, dall'MS-DOS e altri. UNIX considera ciascun file come una sequenza di byte, senza alcuna interpretazione. Questo schema garantisce la massima flessibilità, ma il minimo sostegno.

Un file può essere strutturato in 3 modi differenti:

- Nessuna struttura (sequenza di byte)
- Struttura a Record (sequenza di record, più complessa di una sequenza di byte)
  - record di lunghezza fissa
  - record di lunghezza variabili
- Struttura Complessa
  - grafo/albero di record
  - documento formattato

Le ultime due strutture si possono realizzare tramite la prima struttura (sequenze di byte) mediante appositi caratteri di controllo.

- Dal punto di vista del SO, un file è una sequenza di byte oppure può essere una struttura a record (blocchi) che hanno tutti la stessa lunghezza.
- Dal punto di vista del programma che scriviamo noi, un file può essere una sequenza di byte, una sequenza di record (lunghezza fissa o variabile) oppure una struttura complessa.

Attributi dei file

Nome: identifica in modo univoco il file.

Identificatore: è l'informazione che permette al SO di identificare in modo univoco il file.

Tipo: serve ai sistemi che gestiscono file di tipo diverso. Non si riferisce all'estensione del file, ma al tipo di struttura file da utilizzare per la sua rappresentazione a livello di sistema operativo.

Locazione: è il puntatore al dispositivo e alla locazione del file in tale dispositivo.
Insieme di attributi che ci permettono di capire a quale dispositivo è associato un file; se è un dispositivo di memoria di massa, in quale punto (della memoria di massa) troviamo quell'informazione.

Dimensione: la dimensione del file.

Protezione: sono informazioni di controllo degli accessi al file (ACL).

Ora, data, e identificatore dell'utente: sono informazioni relative alla creazione, all'ultima modifica e all'ultimo uso del file. Servono per la protezione ed il controllo dell'uso.

Esiste un descrittore dei file (**FCB**, File Control Block).

Gli attributi dei file sono conservati nella struttura directory, che risiede a sua volta nella memoria secondaria.
Anche gli attributi dei file di altri dispositivi sono immagazzinati nella struttura delle directory.
La cartella è un contenitore di descrittori di file.

Operazioni sui file

Per definire adeguatamente un file è necessario considerare le operazioni che si possono eseguire su di esso.

- **Create**: per creare un file è necessario compiere due passaggi. In primo luogo si deve trovare lo spazio per il file nel file system. Secondariamente, per il file si deve creare un nuovo elemento nella directory in cui registrare i suoi attributi.
- **Open**: il SO mantiene una piccola tabella contenente informazioni riguardanti tutti i file aperti (tabella dei file aperti). Quando si richiede un'operazione su un file, questo viene individuato tramite un indice in tale tabella, in modo da evitare qualsiasi ricerca. Quando il file non è più attivamente usato viene chiuso dal processo, e il SO rimuove l'elemento a esso associato dalla tabella dei file aperti.
  È un'operazione che riceve il nome del file, lo cerca nella directory, e copia l'elemento ad esso associato in memoria (nella tabella dei file aperti).
  Alcuni SO aprono automaticamente ed implicitamente il file la prima volta

che il processo vi fa riferimento e lo chiudono automaticamente quando il processo che lo ha aperto termina.
Serve per caricare in memoria centrale una serie di informazioni che servono al SO per gestire le richieste di accesso a quel file.

- **Write**: per scrivere in un file è indispensabile una chiamata di sistema che specifichi il nome del file e le informazioni che si vogliono scrivere.
  Dato il nome del file, il sistema cerca la sua posizione nella directory. Il file system deve mantenere un puntatore di scrittura alla locazione nel file in cui deve avvenire l'operazione di scrittura successiva. Il puntatore si deve aggiornare ogni qualvolta si esegue una scrittura.
- **Read**: per leggere da un file è necessaria una chiamata di sistema che specifichi il nome del file e la posizione in memoria dove collocare il successivo blocco del file.
  Anche in questo caso si cerca l'elemento corrispondente nella directory e il sistema deve mantenere un puntatore di lettura alla locazione nel file in cui deve avvenire la successiva operazione di lettura. Una volta completata la lettura, si aggiorna il puntatore. Di solito un processo legge o scrive in un file, e la posizione corrente è mantenuta come un puntatore alla posizione corrente del file.
  Sia le operazioni di lettura sia quelle di scrittura adoperano lo stesso puntatore, risparmiando spazio e riducendo la complessità del sistema.
- **Seek**: si ricerca l'elemento appropriato nella directory e si assegna un nuovo valore al puntatore alla posizione corrente nel file.
  Il riposizionamento non richiede alcuna operazione di I/O.
- **Delete**: si cerca l'elemento relativo al file nella directory, si rilascia lo spazio assegnato al file e si elimina l'elemento di directory.
- **Truncate**: si cancella il contenuto del file mantenendo immutati i suoi attributi, pur azzerando la lunghezza del file e rilasciando lo spazio occupato.
- **Close**: copia l'elemento contenuto nella tabella dei file aperti nella directory.
  In alcuni SO la chiusura avviene implicitamente quando il processo termina.
  Vengono rese permanenti le modifiche apportate al file.

Queste operazioni di base comprendono sicuramente l'insieme minimo delle operazioni richieste per i file. Queste operazioni primitive si possono combinare per compiere altre operazioni.

Tipi di file

Nella progettazione di un file system, ma anche dell'intero SO, si deve sempre considerare la possibilità o meno che quest'ultimo riconosca e gestisca i tipi di file. Un sistema operativo che riconosce il tipo di un file ha la possibilità di trattare il file in modo ragionevole.

file type	usual extension	function
executable	exe, com, bin or none	read to run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rrf, doc	various word-processor formats
library	lib, a, so, dll, mpeg, mov, rm	libraries of routines for programmers
print or view	ps, pdf	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, rm	binary file containing audio or A/V information

Metodi di Accesso

I file memorizzano informazioni; al momento dell'uso è necessario accedere a queste informazioni e trasferirle in memoria.

Esistono due metodi per accedere alle informazioni di un file :
- accesso sequenziale
- accesso diretto

Accesso sequenziale

Il più semplice metodo d'accesso: le informazioni del file si elaborano ordinatamente, un record dopo l'altro.
Vengono letti tutti i byte fino a quando non si raggiunge quello desiderato; allo stesso modo nel caso della scrittura vengono aggiunti record in coda

. Questo metodo però impedisce di modificare un singolo blocco: si è costretti a leggere tutto fino a dove si vuole arrivare e poi si aggiunge la modifica.

Il puntatore di lettura si trova su un determinato carattere all'interno del file; chiedo un'operazione di lettura e leggo quel carattere; l'operazione di lettura mi restituisce quel carattere, mentre il puntatore si sposta sul carattere successivo, e vado avanti così (stessa cosa vale per l'operazione di scrittura, solo che la posizione in cui avviene l'operazione di scrittura diventa l'ultima del file).

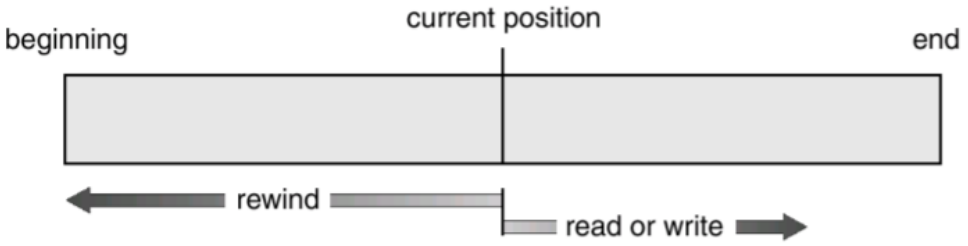
read next: legge alla posizione corrente (cp).

write next: scrive il record nella posizione corrente (il record scritto diventa l'ultimo del file).

reset: riporta la posizione corrente all'inizio del file (cp=0).

Non è possibile fare una *read* dopo una *write* se prima non si fa una *reset*.
Non è possibile fare una
rewrite.

Un'operazione di lettura legge la prima porzione e fa avanzare automaticamente il puntatore del file che tiene traccia della locazione di I/O; analogamente, un'operazione di scrittura fa un'aggiunta in coda al file e avanza fino alla fine delle informazioni appena scritte, che costituisce la nuova fine del file.



sequential access	implementation for direct access
reset	*cp = 0;*
read next	*read cp;* *cp = cp+1;*
write next	*write cp;* *cp = cp+1;*

Accesso diretto

Il file si considera come una sequenza numerata di blocchi o record che si possono leggere o scrivere in modo arbitrario.
Si accede al blocco che ci interessa, e non al singolo byte appartenente al blocco; ecco quindi spiegato il perché si parla di blocchi/porzioni e non sequenze.

read n: legge il blocco n-esimo (cp = n).

write n: scrive il blocco n-esimo (cp = n).

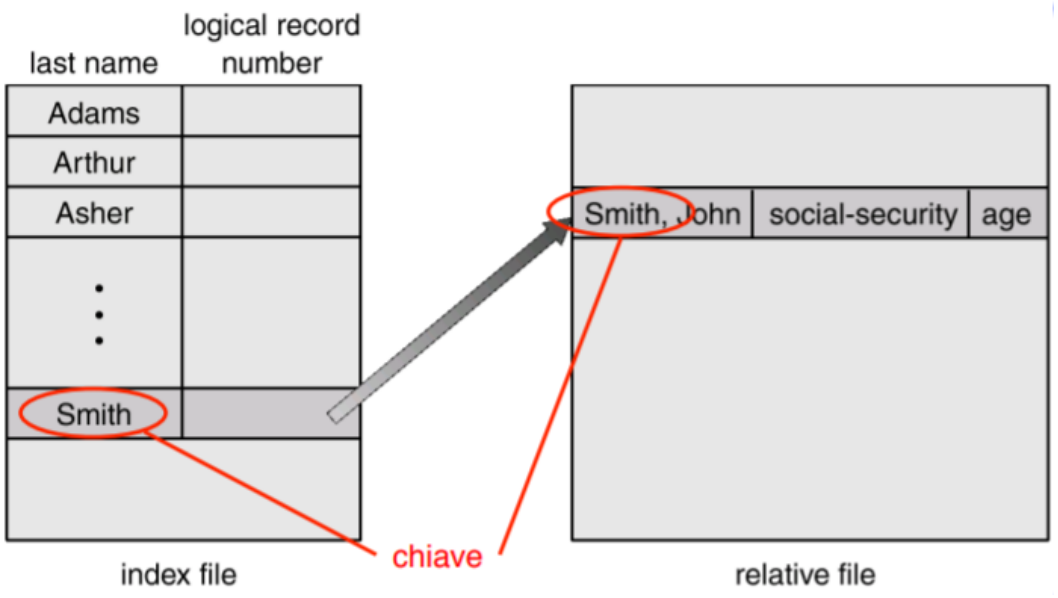
rewrite n: riscrive il blocco n-esimo.

Questa tipologia è alla base di altri metodi di accesso (che si basano sul concetto di chiave): **accesso con indice** e **accesso con hash.**

Accesso con indice

L'indice contiene puntatori ai vari blocchi; per trovare un elemento del file occorre prima cercare nell'indice (tabella degli indici) e quindi usare il puntatore per accedere direttamente al file e trovare l'elemento desiderato. Si associa quindi una chiave ad un indirizzo.

Svantaggi:
- la tabella degli indici occupa memoria;
- ogni accesso ad un file implica una ricerca nella tabella degli indici e nell'accesso vero e proprio.



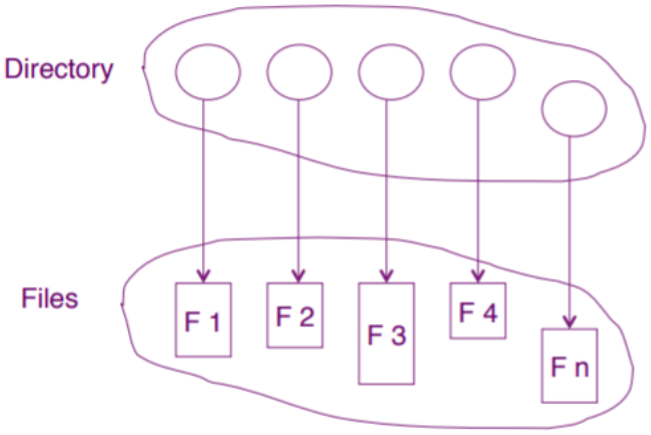
Directory

Introduzione

Una directory è una collezione di file ed è un tipo di dato astratto. È un tipo di file, quindi:
- possiede stessi attributi ed operazioni dei file;
- ha una struttura particolare;
- possiede operazioni aggiuntive.

Un file di tipo directory è una sequenza di elementi di directory; ogni elemento di directory contiene gli attributi di un file che è contenuto dentro quella directory.

Ogni file di tipo directory contiene sempre un elemento di directory che prende il nome di "**..**" e che mi permette di accedere alla cartella genitore (il "**.**" indica la cartella corrente).

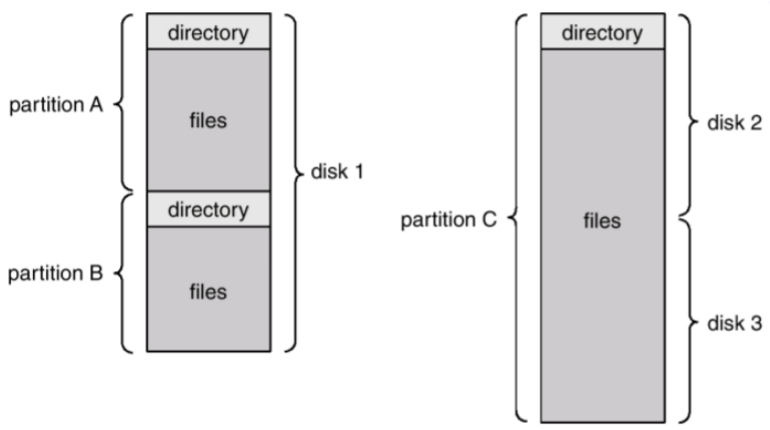


PROBLEMA : per ogni partizione avrei un catalogo con l'elenco di tutti i file, causando così 2 problemi :
• Unicità del nome file.
• La ricerca del file dal punto di vista del kernel risultava complicata.

Si è deciso allora di raggruppare le directory, ovvero in breve ogni utente ha la sua directory (catalogo dei file).

Con il crescere della memoria di massa si ripresentava il problema, quindi si decise che oltre alla directory singola dell'utente esso poteva creare sub-directory dove all'interno memorizzare i diversi file.

Una `open` prende il nome del file, il SO dal nome risale agli attributi, crea un descrittore del file e lo mette in memoria, successivamente useremo il puntatore al descrittore del file.
Come trova il file? Si basa su come è strutturato il SO.



Elemento di directory

Ogni elemento di directory contiene informazione relative ad un file (gli attributi del file) : Nome, Tipo, Locazione, Dimensione attuale, Dimensione massima consentita, Data ultimo accesso, Data ultimo aggiornamento, ID del proprietario e Protezione.

Operazioni sulle directory

Operazioni specifiche sulle directory:

- **Search**: scorrere una directory per individuare l'elemento associato a un particolare file.
- **Create**: creare nuovi file e aggiungerli alla directory.
  Invoca il costruttore di un oggetto di tipo file e va a scrivere nel file di tipo directory per aggiungere gli attributi del file che abbiamo appena creato.
- **Delete**: rimuovere un file dalla directory.
  Corrisponde a due operazioni: da una parte invochiamo il metodo distruttore di un oggetto di tipo file, dall'altra si invoca sempre un'operazione di modifica del file di tipo directory per andare a eliminare un elemento all'interno della cartella stessa.

- **List**: elencare tutti i file di una directory e il contenuto degli elementi della directory associati ai rispettivi file nell'elenco.
- **Rename**: modificare il nome quando il contenuto o l'uso del file subiscono cambiamenti.
- **Traverse** (attraversamento del file system, navigazione): accedere a ogni directory e a ciascun file contenuto in una directory.
  Per motivi di affidabilità, è opportuno salvare il contenuto e la struttura dell'intero file system a intervalli regolari. Questo salvataggio consiste nella copiatura di tutti i file in un nastro magnetico; tale tecnica consente di avere una copia di riserva (backup) che sarebbe utile nel caso in cui si dovesse verificare un guasto nel sistema o più semplicemente se un file non è più in uso.
  In quest'ultimo caso si può liberare lo spazio da esso occupato nel disco, riutilizzabile quindi per altri file.

Organizzazione logica delle directory

Introduzione

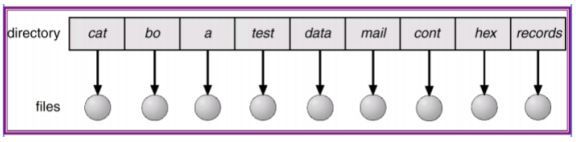
Il file system deve essere organizzato logicamente in modo tale da soddisfare i seguenti requisiti:

- Efficienza: deve essere possibile trovare le informazioni nel modo più veloce possibile.
-
Assegnazione di nomi unici: ogni file deve avere un nome unico, ma l'utente deve avere la possibilità di usare lo stesso nome per file diversi e due nomi diversi per lo stesso file.
-
Raggruppamento: deve essere possibile raggruppare file logicamente affini tra loro, ovvero sulla base delle loro proprietà.

Per soddisfare tali requisiti vengono proposte varie soluzioni.

Directory a livello singolo

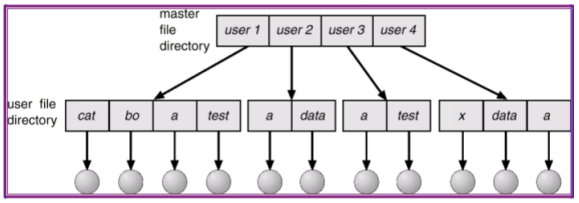
Tutti i file sono contenuti in un'unica directory, comune a tutti gli utenti, facilmente gestibile e comprensibile.



Presenta però limiti notevoli che si manifestano all'aumentare del numero dei file in essa contenuti, oppure se il sistema è usato da più utenti. Poiché si trovano tutti nella stessa directory, i file devono avere nomi unici.

Directory a due livelli

Ogni utente dispone della propria directory utente.



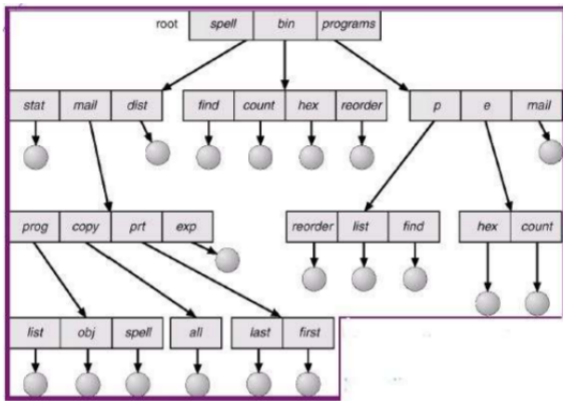
Tutte le directory utente hanno una struttura simile, ma in ciascuna sono elencati solo i file del proprietario.
- Path name.
- Si possono avere file appartenenti ad utenti diversi con lo stesso nome.

- Ricerca più efficiente.

- Ad un utente non è permesso raggruppare i propri file (non può creare directory).

Directory strutturata ad albero

Permette agli utenti di creare proprie sottodirectory e di organizzare i file di conseguenza.
Ogni partizione ha una directory root.
Ogni directory può contenere altre cartelle.



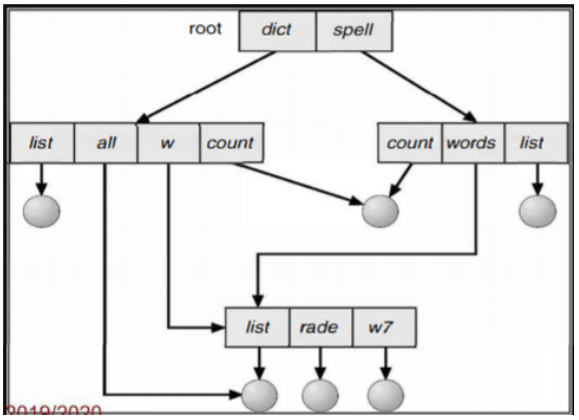
- Ricerca efficiente

- Raggruppamento dei file
- Path name relativo e assoluto

- Directory corrente (directory di lavoro)

Directory strutturata a grafo aciclico

Un grafo aciclico permette alle directory di avere sottodirectory e file condivisi. Lo stesso file o la stessa sottodirectory possono essere in due directory diverse.
Un grafo aciclico, cioè senza cicli, rappresenta la generalizzazione naturale dello schema delle directory con struttura ad albero.

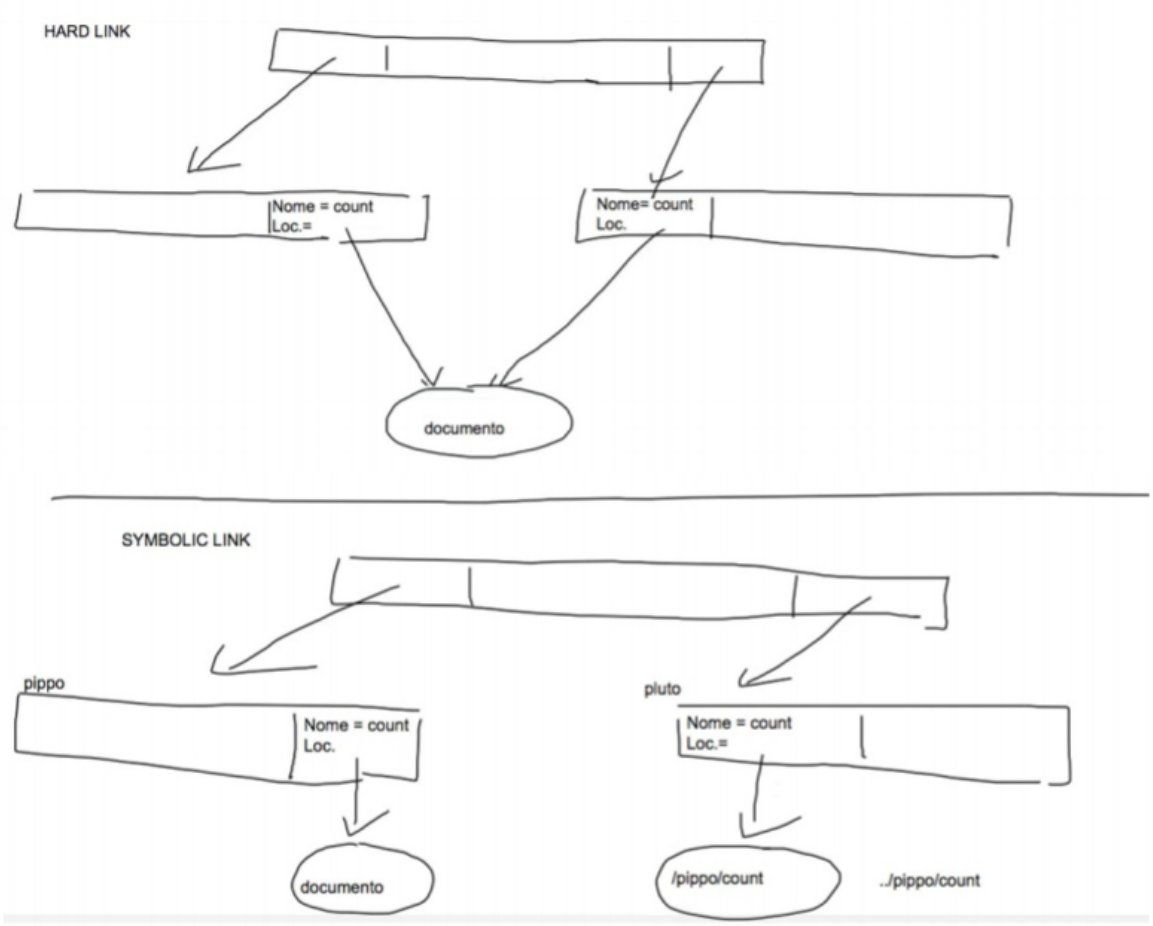


I file e le sottodirectory condivisi si possono realizzare in molti modi. Un metodo diffuso, esemplificato da molti tra i sistemi UNIX, prevede la creazione di un nuovo elemento di directory, chiamato **collegamento**.

Un collegamento (link) è un puntatore a un altro file o a un'altra directory.

- **Collegamenti simbolici** (soft link): lo spazio è reso disponibile solo quando è cancellato il file (MS Windows e Unix).

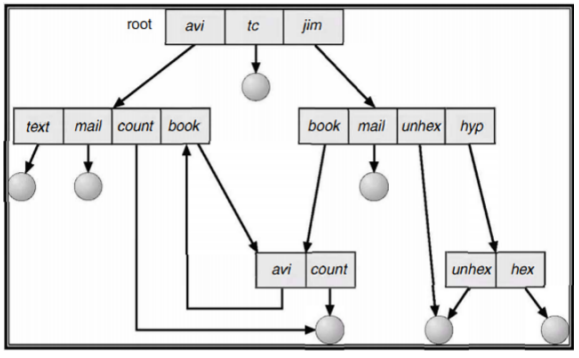
  Sono stringhe che contengono un nome (solitamente relativo) che mi permette di ritrovare quel file. È un file particolare di tipo testo il cui testo è un nome relativo (percorso relativo) o assoluto, che è quello che devo seguire per raggiungere il file che voglio.
- **Collegamenti effettivi** (hard link): lo spazio è reso disponibile quando è cancellato il primo hard link o sono stati cancellati tutti gli hard link.
  È un insieme di informazioni che ci permettono di ritrovare il file, e queste informazioni sono presenti in due elementi di directory distinti che si trovano in due directory distinte.



Windows non permette all'utente di creare collegamenti effettivi, se non il primo (quando viene creato il file), con una sola eccezione rappresentata dalle cartelle.
Linux permette di creare collegamenti effettivi multipli, a patto che non siano a cartelle.

Directory strutturata a grafo generale

É permessa la presenza di cicli.



Se si permette che nella directory esistano cicli, è preferibile evitare una duplice ricerca di un elemento, per motivi di correttezza e di prestazioni.

Bisogna evitare che i processi che

attraversano il file system vadano in loop infinito. Se si è in presenza di un ciclo nel grafo viene impedito al processo il collegamento alla directory.

Windows e Unix permettono la realizzazione di cicli attraverso soft link. Unix non permette hard link a directory.

File System Mounting

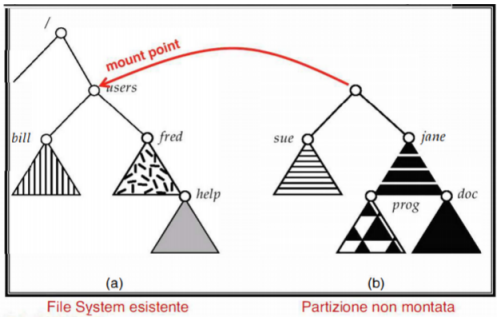
Così come si deve aprire un file per poterlo usare, per essere reso accessibile ai processi di un sistema un file system deve essere montato.

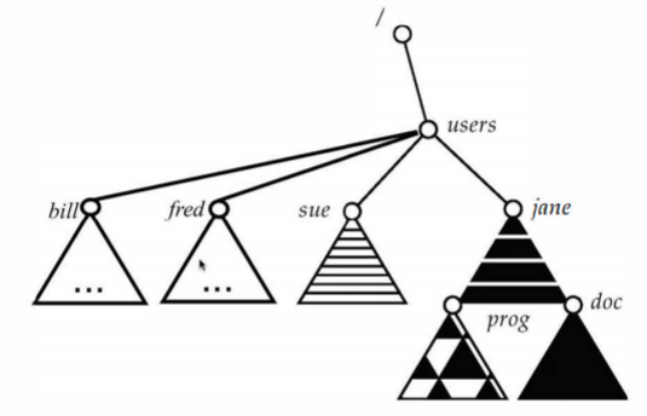
Un file system non montato viene montato ad un **mount point**.

La procedura di montaggio è molto semplice: si fornisce al SO il nome del dispositivo e la sua locazione (mount point = punto di montaggio) nella struttura di file e directory alla quale agganciare il file system.

Di solito, un mount point è una directory vuota a cui sarà agganciato il file system che deve essere montato.
Il passo successivo consiste nella verifica da parte del SO della validità del file system contenuto nel dispositivo.
La verifica si compie chiedendo al driver del dispositivo di leggere la directory di dispositivo e controllando che tale directory abbia il formato previsto.

Infine, il SO annota nella sua struttura della directory che un certo file system è montato al punto di montaggio specificato.
Questo schema permette al sistema operativo di attraversare la sua struttura della directory, passando da un file system all'altro secondo le necessità.





Per illustrare l'operazione di montaggio, si consideri il file system rappresentato nella figura a sinistra, in cui i triangoli rappresentano sottoalberi di directory rilevanti. La parte (a) mostra un file system esistente, mentre nella parte (b) è raffigurato un volume non ancora montato. A questo punto, si può accedere solo ai file del file system esistente.
Nella figura a destra si possono vedere gli effetti dell'operazione di montaggio del volume al punto di montaggio /users. Se si smonta il volume, il file system ritorna alla situazione rappresentata nella figura in alto.

File Speciali

I "file speciali" sono inclusi nella gerarchia delle directory come qualsiasi altro file, ma in realtà corrispondono a devices di I/O.
Possiamo avere file speciali a blocchi (dischi) o a caratteri (tastiere).
Con questo trucco si può fare riferimento ad un dispositivo utilizzando il suo path name.

Protezioni

Se un SO permette l'uso del sistema da parte di più utenti, diventano particolarmente rilevanti i problemi relativi alla condivisione dei file, alla loro identificazione tramite nomi e alla loro protezione.

Il sistema può permettere a ogni utente, in modo predefinito, di accedere ai file degli altri utenti, oppure può richiedere che un utente debba esplicitamente concedere i permessi di accesso ai file.
Per realizzare i meccanismi di condivisione e protezione, il sistema deve memorizzare e gestire più attributi di directory e file rispetto a un sistema che

consente un singolo utente.
La maggior parte dei sistemi ha adottato relativamente a ciascun file o directory, i concetti di proprietario (o utente) e gruppo.

- Il *proprietario* è l'utente che può cambiare gli attributi di un file o directory, concedere l'accesso e che, in generale, ha il maggior controllo sul file o directory.

Il proprietario del file può definire l'esatto insieme di operazioni che i membri del gruppo e gli altri utenti possono eseguire.

- L'attributo di *gruppo* di un file si usa per definire il sottoinsieme di utenti autorizzati a condividere l'accesso al file.

Il problema della protezione comunemente si affronta rendendo l'accesso dipendente dall'identità dell'utente. Lo schema più generale per realizzare gli accessi dipendenti dall'identità consiste nell'associare una lista di controllo degli accessi a ogni file e directory; in tale lista sono specificati i nomi degli utenti e i relativi tipi d'accesso consentiti. Quando un utente richiede un accesso a un file specifico il SO esamina la lista di controllo degli accessi associata a quel file; se tale utente è presente nella lista si autorizza l'accesso, altrimenti si verifica una violazione della protezione e si nega l'accesso al file.

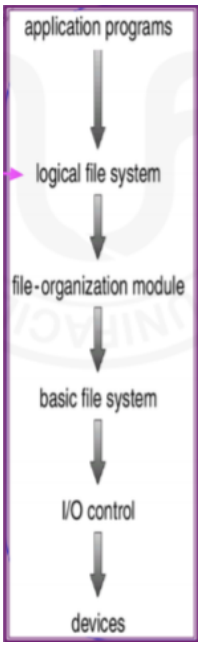
Per condensarne la lunghezza, molti sistemi raggruppano gli utenti di ogni file in tre classi distinte.
-
Proprietario: è l'utente che ha creato il file.
-
Gruppo: si tratta di un insieme di utenti che condividono il file e richiedono tipi di accesso simili.
-
Universo: tutti gli altri utenti del sistema

Realizzazione del File System

File System Stratificato

Lo stesso file system è generalmente composto da molti livelli distinti. Ogni livello si serve delle funzioni dei livelli inferiori per crearne di nuove impiegate

dai livelli superiori. Una system call percorre i diversi strati, a partire dal primo fino all'ultimo, il *driver*, che si occuperà di far svolgere al controller del disco su cui trova il file/directory coinvolto le operazioni necessarie a soddisfare la richiesta.
La struttura stratificata permette di separare gli strati che dipendono dall'hardware dagli strati che sono indipendenti dall'hardware.



Application programs: l'interazione con il file system mediante system call.
Il file è individuato dal suo pathname o dal *FCB handler*. Le librerie di programma trasformano la richiesta in una richiesta di accesso ad un dato blocco logico.

Logical file system: gestisce i metadati, ovvero tutte le strutture del file system (ad eccezione del contenuto dei file). Il file system logico gestisce la struttura della directory per fornire al modulo di organizzazione dei file le informazioni di cui necessita, dato un nome simbolico di file. Mantiene le strutture di file tramite i blocchi di controllo dei file (file control block, FCB), contenenti informazioni sui file come la proprietà, i permessi, e la posizione del contenuto.

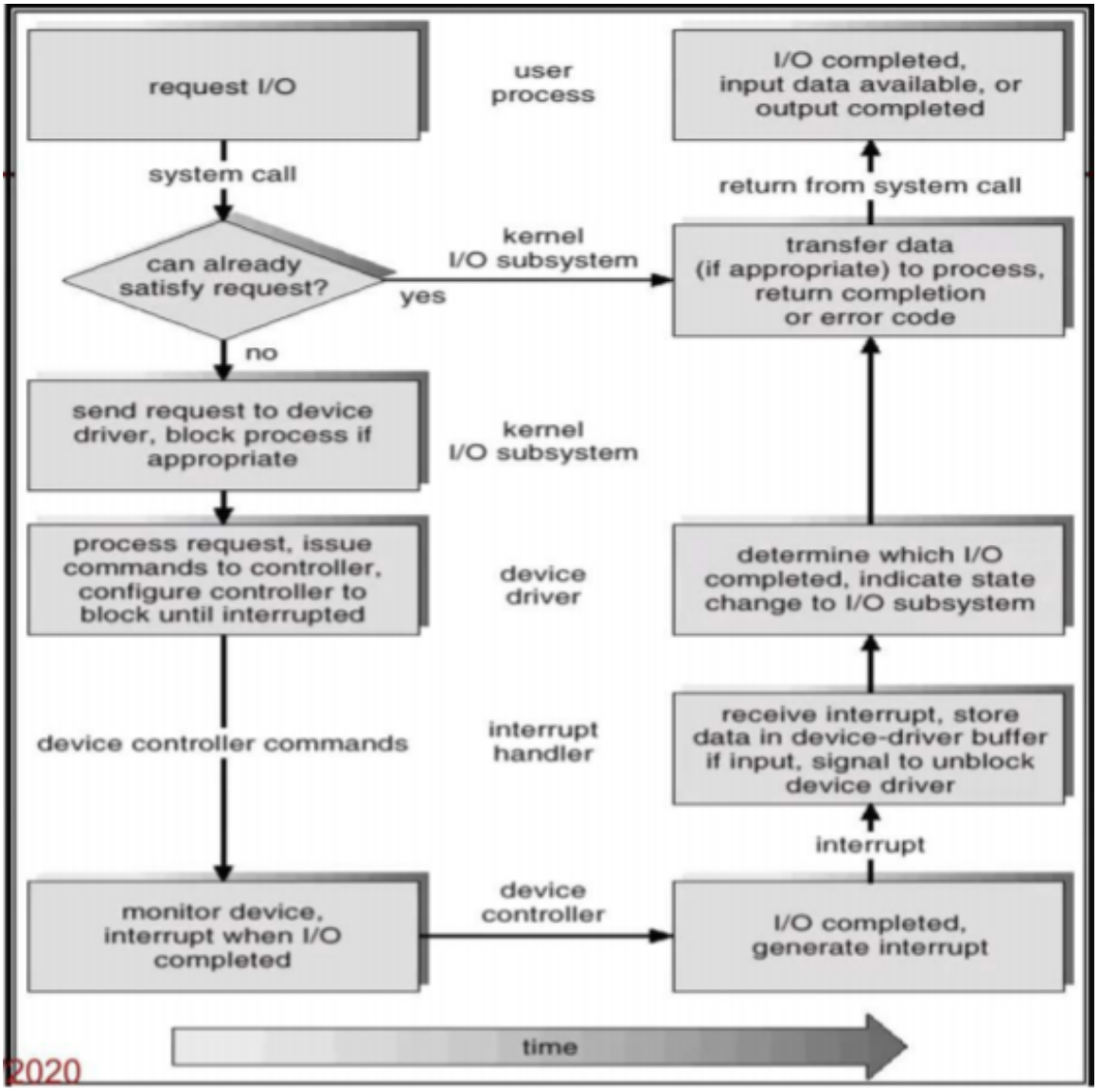
File-organization module: è a conoscenza dei file e dei loro blocchi logici, così come dei blocchi fisici dei dischi. Conoscendo il tipo di allocazione dei file usato e la locazione dei file, può tradurre gli indirizzi dei blocchi logici, che il file system di base deve trasferire, negli indirizzi dei blocchi fisici. Blocchi fisici contenenti tali dati di solito non corrispondono ai numeri dei blocchi logici; per individuare ciascun blocco è quindi necessaria una traduzione.
Il modulo di organizzazione dei file comprende anche il gestore dello spazio libero, che registra i blocchi non assegnati e li mette a disposizione del modulo di organizzazione dei file quando sono richiesti.

Basic file system: caching, buffering, scheduling.
Per i dispositivi di tipo blocco, traduzione dei numeri di blocco in indirizzi del supporto di memorizzazione (*<superficie, cilindro, settore>*).
Comunicazione con i driver per leggere e scrivere blocchi fisici o caratteri.

Deve soltanto inviare dei generici comandi all'appropriato driver di dispositivo per leggere e scrivere blocchi fisici nel disco. Questo strato gestisce inoltre buffer di memoria e le cache che conservano vari blocchi dei file system, delle directory e dei dati. Un blocco viene allocato nel buffer prima che possa verificarsi il trasferimento di un blocco del disco. Quando il buffer è pieno, il gestore del buffer deve recuperare più spazio di memoria per il buffer oppure deve liberare spazio nel buffer per permettere il completamento di un I/O richiesto. Le cache servono a conservare metadati di file system usati frequentemente, in modo da migliorare le prestazioni. La gestione dei loro contenuti è quindi un punto critico per conseguire prestazioni ottimali del sistema.

I/O control: è costituito dai driver dei dispositivi e dai gestori dei segnali d'interruzione; si occupa del trasferimento delle informazioni tra memoria centrale e memoria secondaria.
Un driver di dispositivo si può concepire come un traduttore che riceve comandi ad alto livello, e che emette specifiche istruzioni di basso livello per i dispositivi, usate dal controllore che fa da interfaccia tra i dispositivi di I/O e il resto del sistema. Un driver di dispositivo di solito scrive specifiche configurazioni di bit in specifiche locazioni della memoria del controllore di I/O per indicare quali azioni il dispositivo di I/O debba compiere, e in quali locazioni.

Devices: controller, interazione con i dispositivi.



Virtual File System

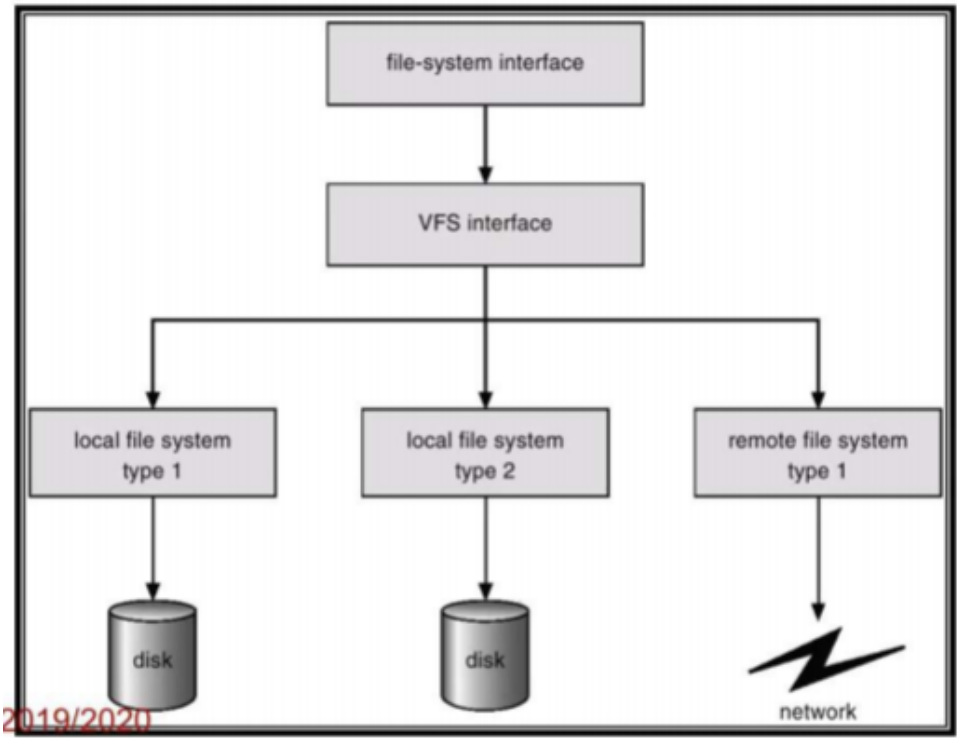
Il VFS fornisce la stessa interfaccia basata su system call (API) da usare per diversi tipi di file system.

La realizzazione del file system si articola in 3 strati principali, riportati in modo schematico nella figura.

- Il primo strato è l'*interfaccia del file system*, basata sulle chiamate di sistema `open()` , `read()` , `write()` e `close ()` e sui descrittori di file.
- Il secondo strato si chiama strato del *file system virtuale* (virtual file system, VFS) e svolge due funzioni importanti:
  1) Separa le operazioni generiche del file system dalla loro realizzazione definendo un'interfaccia VFS uniforme. Nello stesso calcolatore possono coesistere più interfacce VFS, che permettono un accesso trasparente a

diversi tipi di file system montati localmente.
2) Permette la rappresentazione univoca di un file su tutta la rete. Il VFS è basato su una struttura di rappresentazione dei file detta *v-node* che contiene un indicatore numerico unico per tutta la rete per ciascun file (gli i-node di UNIX sono unici solo all'interno di un singolo file system). Tale unicità per tutta la rete è richiesta per la gestione del file system di rete.

- Il terzo strato è il *file system locale.*



Logical file system

Logical file system: gestisce i metadati, ovvero tutte le strutture del file system (ad eccezione del contenuto dei file).
Il file system logico gestisce la struttura della directory per fornire al modulo di organizzazione dei file le informazioni di cui necessita, dato un nome simbolico di file. Mantiene le strutture di file tramite i blocchi di controllo dei file (file control block, FCB), contenenti informazioni sui file come la proprietà, i permessi, e la posizione del contenuto.

Strutture del File System in Memoria

Il kernel mantiene le informazioni relative allo stato dei dispositivi di I/O; per esempio la tabella dei file aperti, la tabella delle connessioni di rete, lo stato dei

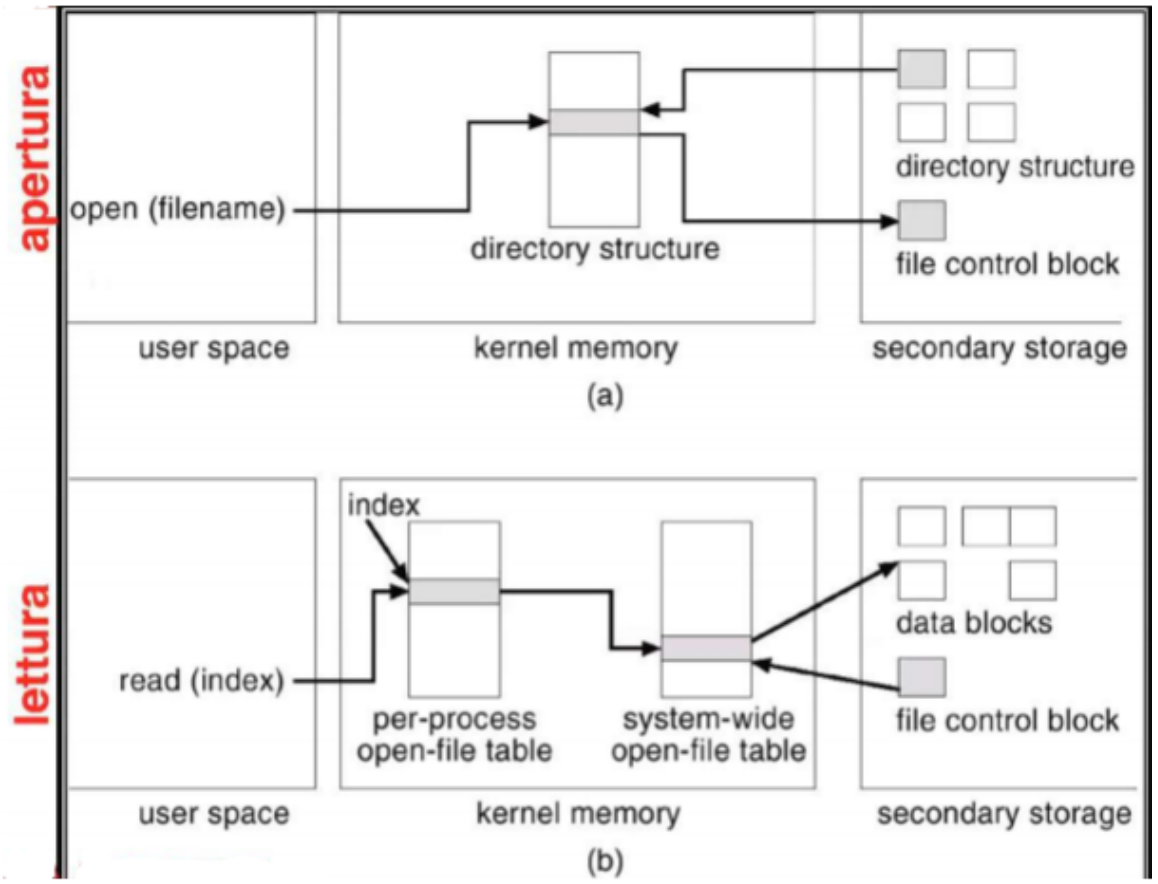
dispositivi a carattere.
In alcuni SO l'I/O viene realizzato con il paradigma ad oggetti e lo scambio dei messaggi. Più semplice da progettare ma meno efficiente.

Descrittore dei file (File Control Block, FCB):
- File permissions
- File dates (create, access, write)
- File owner, group, ACL
- File size
- File data blocks

Le applicazioni, per creare un nuovo file, eseguono una chiamata al file system logico, il quale conosce il formato della struttura della directory. Esso crea e alloca un nuovo FCB. Il sistema carica quindi la directory appropriata in memoria, la aggiorna con il nome del nuovo file e con il FCB associato, e la scrive nuovamente sul disco.

Alcuni SO trattano le directory esattamente come i file, distinguendole con un campo per il tipo che indica che si tratta di una directory. Altri dispongono di chiamate di sistema distinte per i file e le directory e trattano le directory come entità separate dai file.

Indipendentemente da questioni strutturali, il file system logico può basarsi sul modulo che si occupa dell'organizzazione dei file per far corrispondere l'I/O su directory a numeri di blocchi di disco, che poi si passano al file system di base e al sistema per il controllo dell'I/O. Una volta creato un file, per essere usato per operazioni di I/O deve essere aperto.



La chiamata di sistema `open()` passa un nome di file al file system. Per controllare se il file sia già in uso da parte di qualche processo, la chiamata `open()` dapprima esamina la tabella dei file aperti in tutto il sistema; in caso affermativo, aggiunge un elemento alla tabella dei file aperti del processo che punta alla tabella dei file aperti in tutto il sistema.

L'algoritmo può eliminare significativi rallentamenti.

Una volta aperto il file, se ne ricerca il nome all'interno della directory. Alcune porzioni della struttura delle directory sono di solito tenute in memoria per accelerare le operazioni sulle directory. Una volta trovato il file, si copia l'FCB nella tabella generale dei file aperti, tenuta in memoria. Questa tabella non solo contiene l'FCB, ma tiene anche traccia del numero di processi che in quel momento hanno il file aperto.
Successivamente, si crea un elemento nella tabella dei file aperti del processo con un puntatore alla tabella generale e con alcuni altri campi.

Questi altri campi possono comprendere un puntatore alla posizione corrente nel file (per successive operazioni `read()` o `write()`) e il tipo d'accesso richiesto all'apertura del file.

La `open()` riporta un puntatore all'elemento appropriato nella tabella dei file aperti del processo, sicché tutte le operazioni sul file si svolgeranno usando questo puntatore. Finché un file non viene chiuso, tutte le operazioni si compiono sulla tabella dei file aperti usando questo elemento.

Realizzazione delle Directory

La selezione degli algoritmi di allocazione e degli algoritmi di gestione delle directory ha un grande effetto sull'efficienza, le prestazioni e l'affidabilità del file system. Per tale ragione è necessario comprendere i vari aspetti di questi algoritmi.

- **Lista lineare**: è basato sull'uso di una lista lineare contenente i nomi dei file con puntatori ai blocchi di dati.

  - Ricerca: lineare.
  - Aggiunta: ricerca lineare e, se non esistono omonimi, aggiunta in coda.
  - Cancellazione: ricerca lineare, marcatura dell'elemento come libero, oppure si copia l'ultimo elemento in quello liberato oppure si usa una lista concatenata.

  Questo metodo è di facile programmazione, ma la sua esecuzione è onerosa in termini di tempo.

- **Hash Table**: in questo metodo una lista lineare contiene gli elementi di directory, ma si usa anche una struttura dati hash. La tabella hash riceve un valore calcolato usando come operando il nome del file e riporta un puntatore al nome del file nella lista lineare.

  - Tempo di ricerca molto breve.
  - Bisogna gestire le collisioni.
  - La dimensione della tabella è fissa e decisa a priori.

  Questa struttura dati può diminuire notevolmente il tempo di ricerca nella directory.

File-organization module

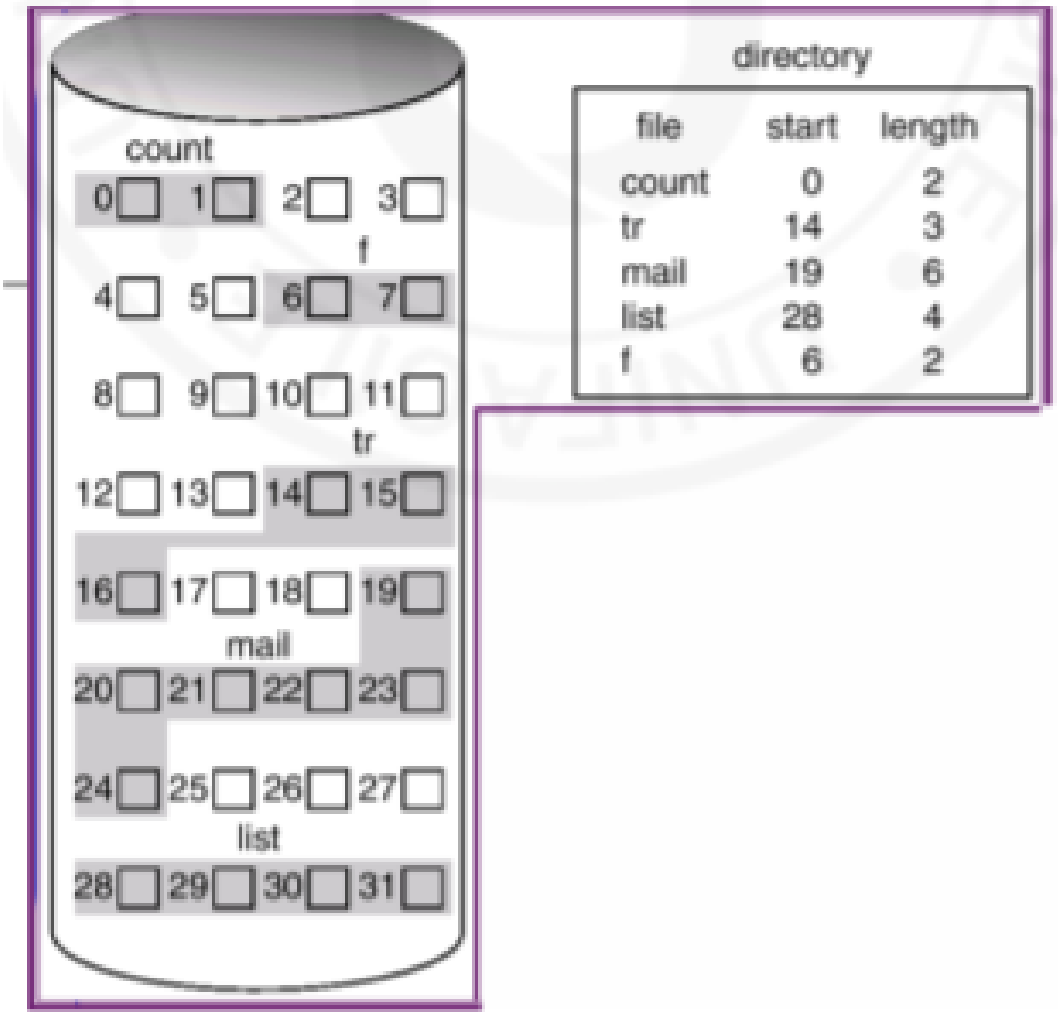
File-organization module: è a conoscenza dei file e dei loro blocchi logici, così come dei blocchi fisici dei dischi. Conoscendo il tipo di allocazione dei file usato e la locazione dei file, può tradurre gli indirizzi dei blocchi logici, che il file system di base deve trasferire, negli indirizzi dei blocchi fisici. Blocchi fisici contenenti tali dati di solito non corrispondono ai numeri dei blocchi logici; per individuare ciascun blocco è quindi necessaria una traduzione.
Il modulo di organizzazione dei file comprende anche il gestore dello spazio libero, che registra i blocchi non assegnati e li mette a disposizione del modulo di organizzazione dei file quando sono richiesti.

Metodi di allocazione

Una partizione è suddivisa in blocchi fisici, che hanno la stessa dimensione dei blocchi logici. Si usano le stesse tecniche di allocazione della memoria centrale.
La natura ad accesso diretto dei dischi permette una certa flessibilità nella realizzazione dei file
. In quasi tutti i casi, molti file si memorizzano nello stesso disco. Il problema principale consiste dunque nell'allocare lo spazio per questi file in modo che lo spazio nel disco sia usato efficientemente e l'accesso ai file sia rapido.
Esistono 3 metodi principali per l'allocazione dello spazio di un disco:
1) ALLOCAZIONE CONTIGUA
2) ALLOCAZIONE CONCATENATA
3) ALLOCAZIONE INDICIZZATA

1) Allocazione contigua

Ogni file occupa un insieme di blocchi contigui nel disco.



Gli indirizzi del disco definiscono un ordinamento lineare nel disco stesso.

L'allocazione contigua dello spazio per un file è definita dall'indirizzo del primo blocco (inteso come numero di blocco, *start*) e dalla lunghezza (espressa in blocchi, *length*). Se il file è lungo n blocchi e comincia dalla locazione b, allora occupa i blocchi $b, b+1, b+2, ..., b+n-1$.

L'elemento di directory per ciascun file indica l'indirizzo del blocco d'inizio e la lunghezza dell'area assegnata per questo file.

Consente l'accesso diretto ed è facilmente implementabile.
Accedere a un file il cui spazio è assegnato in modo contiguo è facile.

Quando si usa un accesso sequenziale, il file system memorizza l'indirizzo dell'ultimo blocco cui è stato fatto riferimento e, se necessario, legge il blocco

successivo.

Nel caso di un accesso diretto al blocco i di un file che comincia al blocco b, si può accedere immediatamente al blocco $b + i$.
Quindi, sia l'accesso sequenziale sia quello diretto si possono gestire con l'allocazione contigua.

L'allocazione contigua presenta però alcuni problemi (gli stessi dell'allocazione dinamica della memoria).

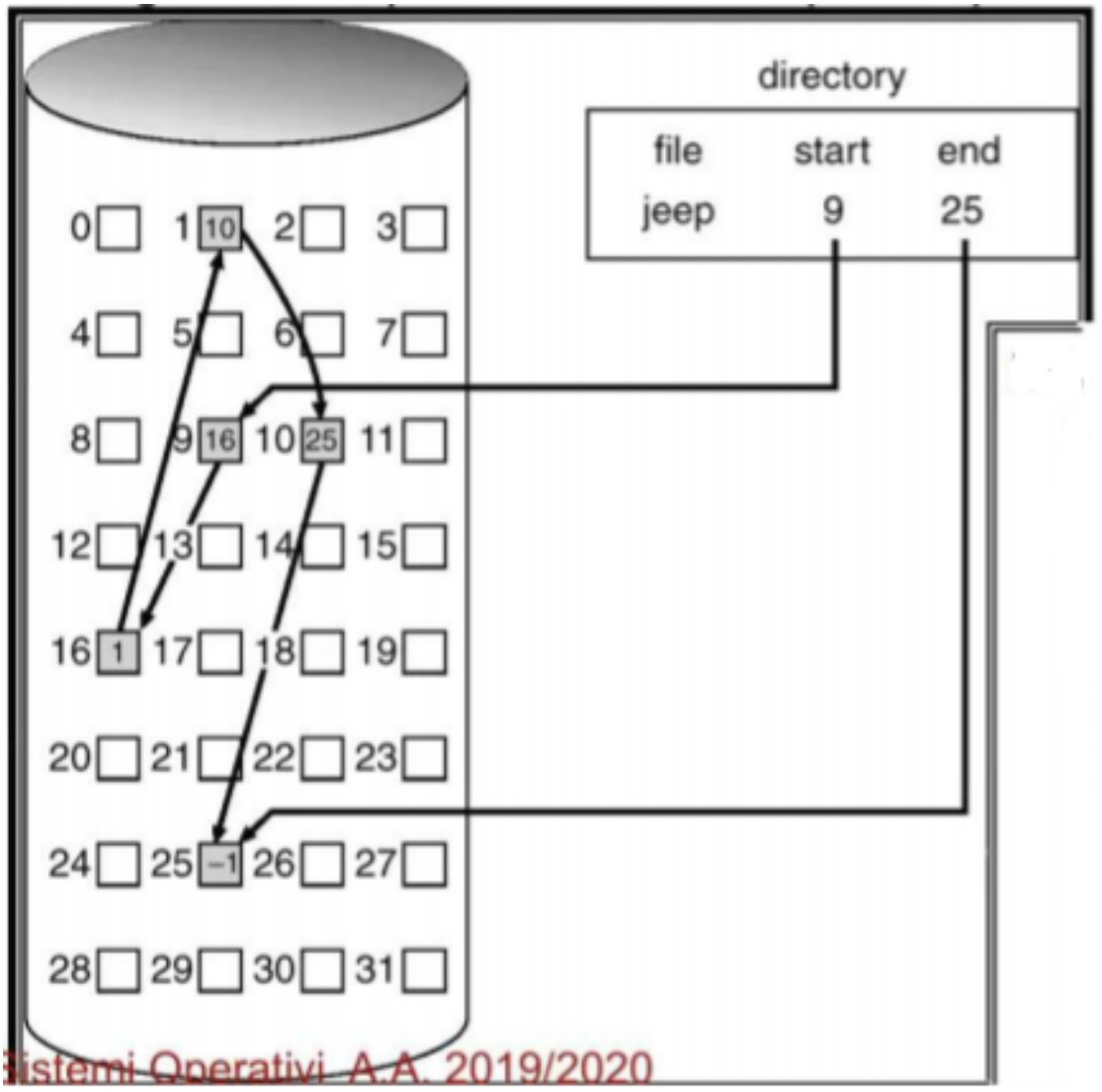
Un problema è, infatti, quello di soddisfare una richiesta di dimensione n data una lista di buchi liberi. I più comuni criteri di scelta di un buco libero da un insieme di buchi disponibili sono il first-fit e il best-fit. Questi algoritmi soffrono della frammentazione esterna.
Un altro problema che riguarda l'allocazione contigua è la determinazione della quantità di spazio necessaria per un file. Quando si crea un file, occorre trovare e allocare lo spazio di cui necessita.
Esiste il problema di conoscere la dimensione del file da creare; in alcuni casi questa dimensione si può stabilire in modo abbastanza semplice, ad esempio quando si copia un file esistente. In generale, tuttavia, non è facile stimare la dimensione di un file che deve contenere dati emessi da un programma.
Anche se si conosce in anticipo la quantità di spazio necessaria per un file, l'allocazione preventiva può in ogni modo essere inefficiente.

Il file ha perciò un'estesa frammentazione interna.

Per ridurre al minimo questi inconvenienti alcuni SO fanno uso di uno schema di allocazione continua modificato, chiamato **Workaround** : inizialmente si assegna una porzione di spazio contiguo, e se questa non è abbastanza grande si aggiunge un'altra porzione di spazio (estensione). La locazione dei blocchi dei file si registra come la locazione di partenza e il numero dei blocchi, insieme con l'indirizzo del primo blocco dell'estensione seguente.

2) Allocazione concatenata

Ogni file è una lista concatenata di blocchi (ogni blocco conterrà le informazioni relative a se stesso più il puntatore al blocco logico successivo). Ogni blocco può trovarsi in un punto qualsiasi del disco.



Risolve tutti i problemi dell'allocazione contigua.
Con questo tipo di allocazione, infatti, ogni file è composto da una lista concatenata di blocchi del disco i quali possono essere sparsi in qualsiasi punto del disco stesso. La directory contiene un puntatore al primo e all'ultimo blocco del file.
In ogni blocco fisico si mette il puntatore al blocco fisico successivo.
Con questa soluzione non si hanno problemi di frammentazione esterna e resta di facile implementazione.
Non è necessario stimare la dimensione massima del file alla sua creazione. Per creare un nuovo file si crea semplicemente un nuovo elemento nella directory.
Con l'allocazione concatenata, ogni elemento della directory ha un puntatore al primo blocco del file. Questo puntatore si inizializza a
null (valore del puntatore di fine lista) a indicare un file vuoto; anche il campo

della dimensione s'imposta a 0.
Per leggere un file occorre semplicemente leggere i blocchi seguendo i puntatori da un blocco all'altro.
Con l'allocazione concatenata non esiste frammentazione esterna e per soddisfare una richiesta si può usare qualsiasi blocco libero della lista. Inoltre non è necessario dichiarare la dimensione di un file al momento della sua creazione.
Un file può continuare a crescere finché sono disponibili blocchi liberi, di conseguenza non è mai necessario compattare lo spazio del disco.

L'allocazione concatenata presenta comunque alcuni svantaggi.

Il problema principale riguarda il fatto che può essere usata in modo efficiente solo per i file ad accesso sequenziale.

Per trovare l'i-esimo blocco di un file occorre partire dall'inizio del file e seguire i puntatori finché non si raggiunge l'i-esimo blocco. Ogni accesso a un puntatore implica una lettura del disco, e talvolta un posizionamento della testina.
Di conseguenza, per file il cui spazio è assegnato in modo concatenato, la funzione d'accesso diretto è inefficiente.
Un altro svantaggio dell'allocazione concatenata riguarda lo spazio richiesto per i puntatori. Ogni file richiede un po' più spazio di quanto ne richiederebbe altrimenti. La soluzione più comune a questo problema consiste nel riunire un certo numero di blocchi contigui in *cluster* (gruppi di blocchi), e nell'allocare i cluster anziche i blocchi.

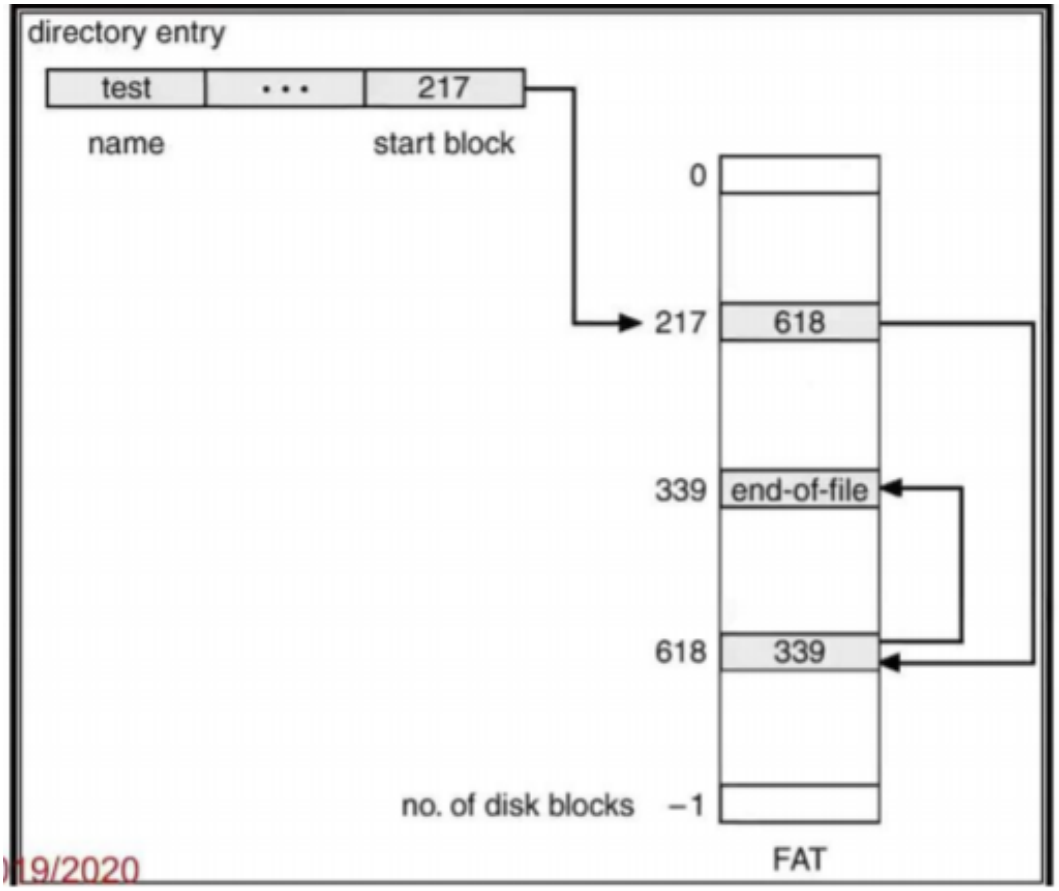
Un altro problema riguarda l'affidabilità. Poiché i file sono tenuti insieme da puntatori sparsi per tutto il disco, si immagini che cosa accadrebbe se un puntatore andasse perduto o danneggiato. Un errore di programmazione del SO oppure un errore di un'unità a disco potrebbero causare il prelevamento del puntatore errato. Questo errore, a sua volta, potrebbe causare il collegamento alla lista dei blocchi liberi oppure a un altro file. *Una soluzione parziale a tale problema consiste nell'usare liste doppiamente concatenate oppure nel memorizzare il nome del file e il relativo numero di blocco in ogni blocco*; questi schemi però sono ancora più onerosi per ogni file.

Traduzione indirizzo:

- LA : indirizzo logico
-
B : numero di blocco
-
S : scostamento

$\rightarrow B = LA/511$
$\rightarrow$
$S = (LA \ \% \ 511) + 1$

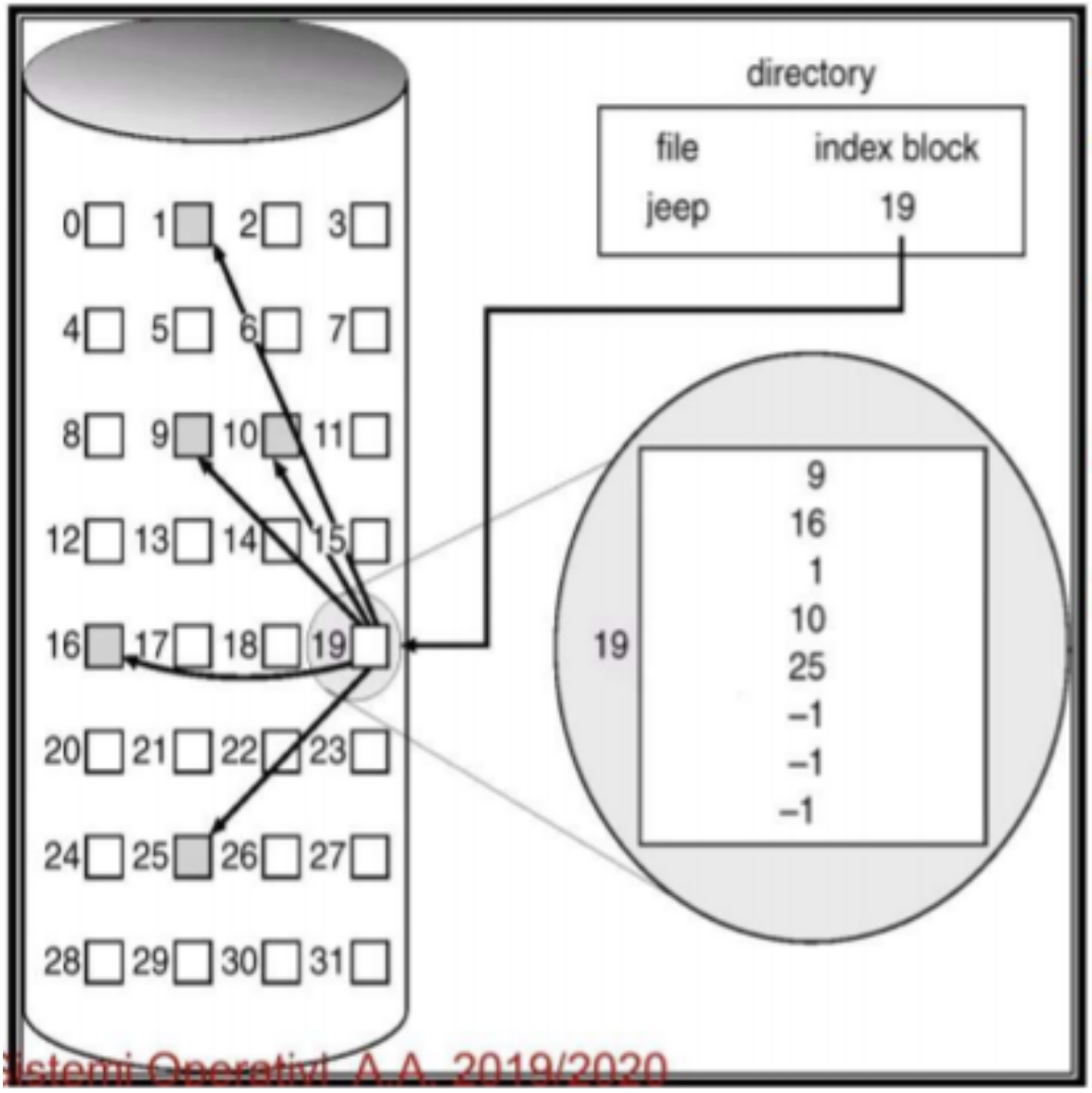
FILE ALLOCATION TABLE (**FAT**): la FAT consiste in una variante importante del metodo di allocazione concatenata.
Per contenere tale tabella si riserva una sezione del disco all'inizio di ciascun volume. La FAT ha un elemento per ogni blocco del disco ed è indicizzata dal numero di blocco; si usa essenzialmente come una lista concatenata.
L'elemento di directory contiene il numero del primo blocco del file. L'elemento della tabella indicizzato da quel numero di blocco contiene a sua volta il numero del blocco successivo del file. Questa catena continua fino all'ultimo blocco, che ha come elemento della tabella un valore speciale di fine del file
. I blocchi inutilizzati sono indicati nella tabella da un valore 0. L'allocazione di un nuovo blocco a un file implica semplicemente la localizzazione del primo elemento della tabella con valore 0 e la sostituzione del valore di fine del file precedente con l'indirizzo del nuovo blocco; lo 0 è quindi sostituito con il valore di fine del file.
Si deve garantire l'affidabilità, perché perdendo il blocco della FAT si perdono tutti i file.



3) Allocazione indicizzata

In mancanza di una FAT, l'allocazione concatenata non è in grado di sostenere un efficiente accesso diretto, poiché i puntatori ai blocchi sono sparsi, con i blocchi stessi, per tutto il disco e si devono recuperare in ordine.

L'allocazione indicizzata risolve questo problema, raggruppando tutti i puntatori in una sola locazione, nella tabella indice.



Si tratta di un array d'indirizzi di blocchi del disco. L'i-esimo elemento della tabella indice punta all'i-esimo blocco del file. La directory contiene l'indirizzo della tabella indice.
Il FCB contiene l'indirizzo del blocco fisico che occupa la tabella indice.

Vantaggi:
- È garantito l'
accesso diretto.
- Accesso dinamico
senza frammentazione esterna.

- File di lunghezza illimitata.

Svantaggi:
- Necessita di riscrivere spazio per la tabella indice
- Sovraccarico per la gestione della tabella indice.

Anche in questo caso alla chiusura di un file bisogna ricopiare la tabella indice sul disco.

Chiudere un file significa rimuovere il FCB e rimuovere la tabella indice, previa copia sul disco delle modiche fatte. Il problema nasce quando vado a smontare il disco: quindi ecco spiegato perché devo fare “rimuovi dispositivo”, così ho la certezza che le modiche effettuate sono state copiate.

Schema concatenato

La tabella degli indici è realizzata con una lista concatenata di blocchi.

Traduzione degli indirizzi:
-
LA : indirizzo logico
-
Q_1 : blocco della tabella indice
-
Q_2 : scostamento nel blocco della tabella indice
-
R_2 : scostamento nel blocco del file
→ $Q_1 = LA/(512 * 511)$
→
$R_1 = LA \ \% \ (512 * 511)$
→
$Q_2 = R_1/512$
→
$R_2 = R_1 \ \% \ 512$

Indice a più livelli

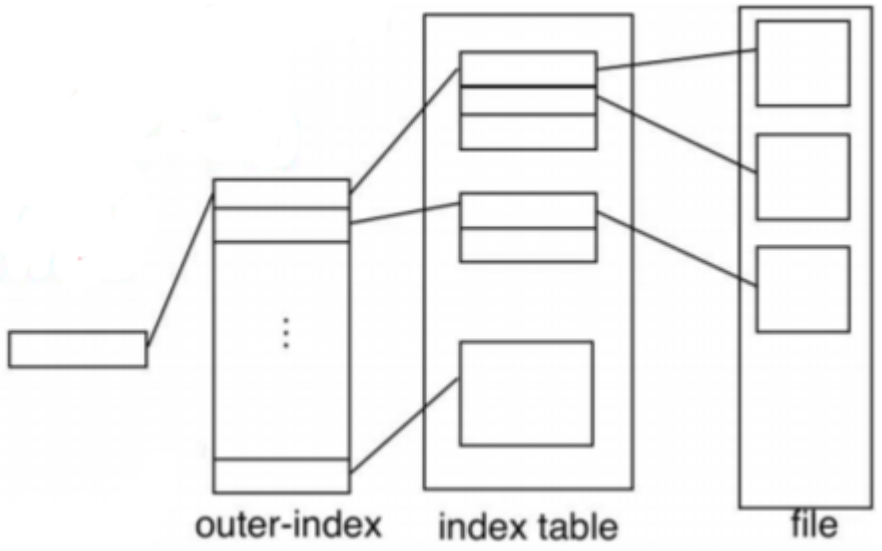
Una variante della rappresentazione concatenata consiste nell’impiego di un blocco indice di primo livello che punta a un insieme di blocchi indice di secondo livello che, a loro volta, puntano ai blocchi dei file.

Si organizza quindi la tabella indice in più livelli per gestire file di grandi dimensioni.
Il problema è che per accedere al blocco di interesse, bisogna attraversare tutti i livelli (accessi sul disco) delle tabelle degli indici.

Se tali accessi sono giustificabili per i file di grandi dimensioni, non lo sono per file di piccole dimensioni.

La tabella degli indici è a sua volta indicizzata.

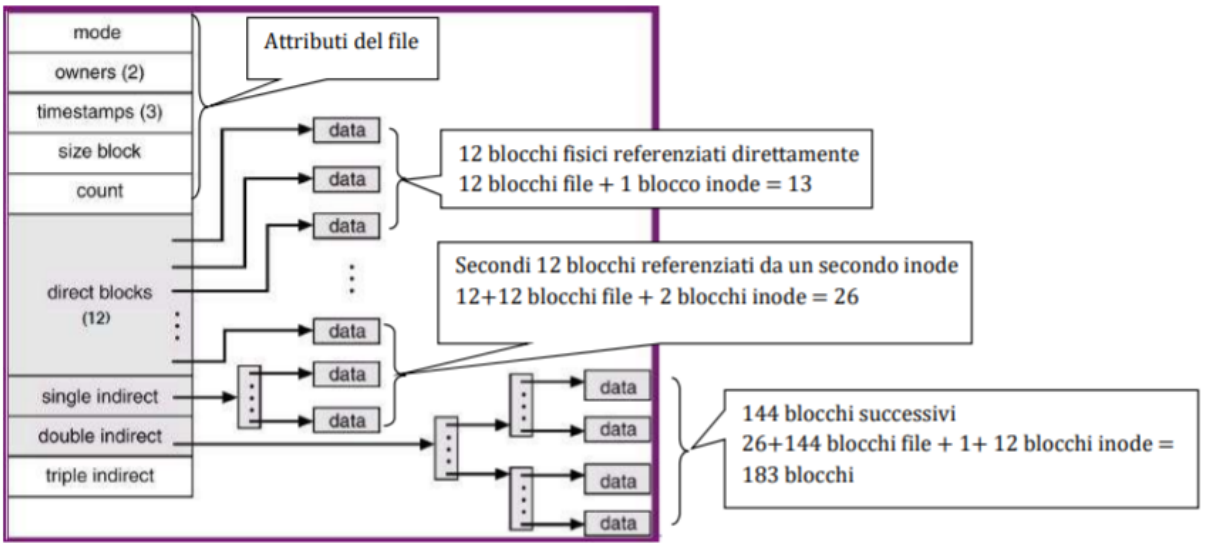
Traduzione degli indirizzi (esempio a 2 livelli) :
-
LA : indirizzo logico
-
Q_1 : scostamento nella tabella indice di 1° livello (outer index)
-
Q_2 : scostamento nel blocco della tabella indice di 2° livello
-
R_2 : scostamento nel blocco del file
$\rightarrow Q_1 = LA/(512 * 512)$
→
$R_1 = LA \ \% \ (512 * 512)$
→
$Q_2 = R_1/512$
→
$R_2 = R_1 \ \% \ 512$



Schema combinato

Un'altra possibilità è la soluzione adottata nell'UFS *(file system Unix)*, consistente nel tenere i primi 15 puntatori del blocco indice nell'i-node del file. 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 per 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 puntatore di blocco indiretto è l'indirizzo di un blocco indiretto singolo; si tratta di un blocco indice che non contiene dati, ma indirizzi di blocchi che contengono dati. Poi c'è un puntatore di 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. L'ultimo puntatore contiene l'indirizzo di un blocco indiretto triplo.
Con questo metodo, il numero dei blocchi che si può allocare a un file supera la quantità di spazio che possono indirizzare i puntatori a file di 4 byte usati da molti SO.

Si ottimizzano gli accessi multipli alla tabella degli indici, adattando i diversi livelli di reindirizzamento.



Gestione dello spazio libero

Poiché la quantità di spazio dei dischi è limitata, è necessario riutilizzare lo spazio lasciato dai file cancellati per scrivere, se è possibile, nuovi file.
Per tener traccia dello spazio libero in un disco, il sistema conserva una lista dello spazio libero; vi sono registrati tutti gli spazi liberi, cioè non allocati ad alcun file o directory. Per creare un file occorre cercare nella lista dello spazio libero la quantità di spazio necessaria e assegnarla al nuovo file, quindi rimuovere questo spazio dalla lista.
Quando si cancella un file, si aggiungono alla lista dello spazio libero i blocchi di disco a esso assegnati.

Vettore di bit → spesso la lista dello spazio libero si realizza come una mappa di bit, o vettore di bit. Ogni blocco è rappresentato da un bit: se il blocco è libero il bit è 0, se il blocco è assegnato il bit è 1.

$$bit[i] = \begin{cases} 0 \Rightarrow block[i] \text{ libero} \\ 1 \Rightarrow block[i] \text{ occupato} \end{cases}$$

n° blocco = (# bit per parola) * (# parole di valore 0) + (offset del primo bit a 1)

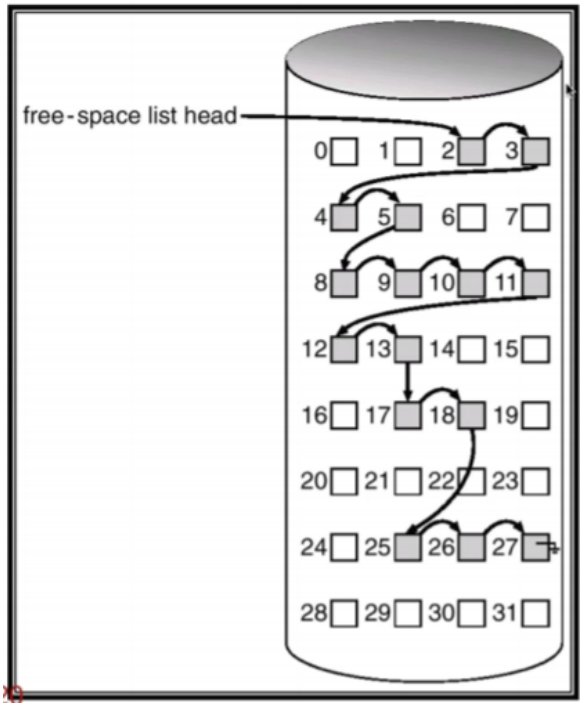
I vantaggi principali che derivano da questo metodo sono la sua relativa semplicità ed efficienza nel trovare il primo blocco libero o n blocchi liberi consecutivi nel disco.
Sfortunatamente, i vettori di bit sono efficienti solo se tutto il vettore è mantenuto in memoria centrale, e viene di tanto in tanto scritto in memoria secondaria allo scopo di consentire eventuali operazioni di ripristino; è

possibile tenere il vettore in memoria centrale se i dischi sono piccoli, come quelli usati nei microcalcolatori; tale soluzione non è applicabile ai dischi più grandi.

Vantaggi:
- Facile da realizzare
- Efficiente se tenuta in memoria centrale
- Utilizzabile solamente con dei dischi di piccole dimensioni

LISTA CONCATENATA DI BLOCCHI LIBERI

Un altro metodo di gestione degli spazi liberi consiste nel collegarli tutti, tenere un puntatore al primo di questi in una speciale locazione del disco e caricarlo in memoria. Questo primo blocco contiene un puntatore al successivo blocco libero, e così via.



Basic file system

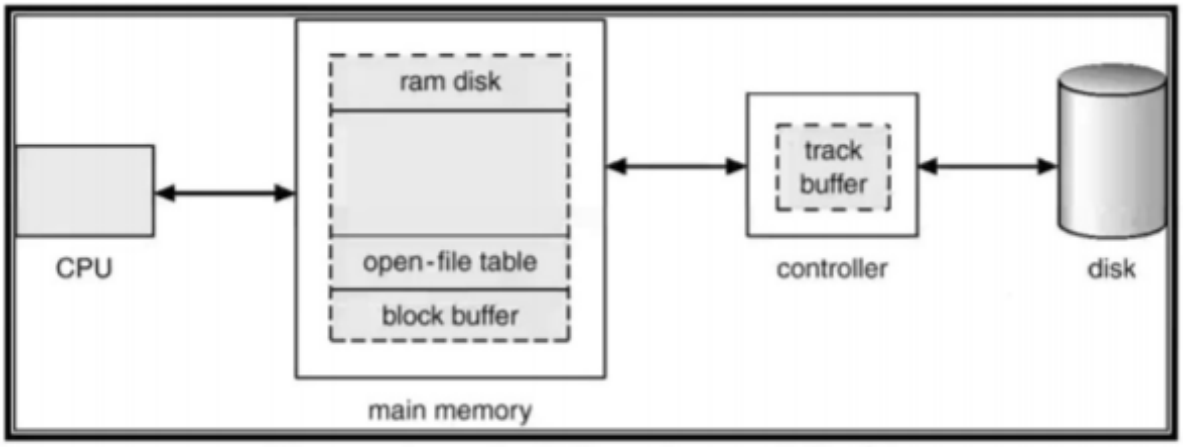
Basic file system: caching, buffering, scheduling.
Per i dispositivi di tipo blocco, traduzione dei numeri di blocco in indirizzi del supporto di memorizzazione (
<superficie, cilindro, settore>).
Comunicazione con i driver per leggere e scrivere blocchi fisici o caratteri.
Deve soltanto inviare dei generici comandi all'appropriato driver di dispositivo

per leggere e scrivere blocchi fisici nel disco. Questo strato gestisce inoltre buffer di memoria e le cache che conservano vari blocchi dei file system, delle directory e dei dati. Un blocco viene allocato nel buffer prima che possa verificarsi il trasferimento di un blocco del disco. Quando il buffer è pieno, il gestore del buffer deve recuperare più spazio di memoria per il buffer oppure deve liberare spazio nel buffer per permettere il completamento di un I/O richiesto. Le cache servono a conservare metadati di file system usati frequentemente, in modo da migliorare le prestazioni. La gestione dei loro contenuti è quindi un punto critico per conseguire prestazioni ottimali del sistema.

Cache del disco

Per migliorare le prestazioni del file system si può utilizzare una parte della memoria centrale come cache del disco. Nella cache del disco vengono tenuti i blocchi che si prevede di utilizzare entro breve tempo.
Per migliorare le prestazioni nei PC (personal computer), a volte una parte della memoria viene utilizzata come
disco virtuale (RAM disk).

I migliori algoritmi di sostituzione dei blocchi sono LRU e LFU: il primo si basa sulle pagine recenti e il secondo sulla frequenza di utilizzo delle pagine.



Diverse locazioni di cache per i dischi

Cache delle pagine

Altri sistemi impiegano una cache delle pagine per i file; si tratta di una soluzione che impiega tecniche di memoria virtuale per la gestione dei dati dei file come pagine anziché come blocchi di file system.
L'uso degli indirizzi virtuali è molto più efficiente dell'uso dei blocchi fisici di

disco.
Questo permette di utilizzare la cache sia per le pagine dei dati del disco sia per le pagine dei processi; questo metodo è noto come *memoria virtuale unificata*.

Gli algoritmi di sostituzione delle pagine utilizzati sono LRU, LFU più la paginazione con priorità; in quest'ultimo caso viene data priorità alle pagine dei processi rispetto a quelle dei dati del disco.

Un blocco fisico in cache del disco viene gestito come una pagina fisica di memoria e rientra nello spazio di memoria virtuale unificato.

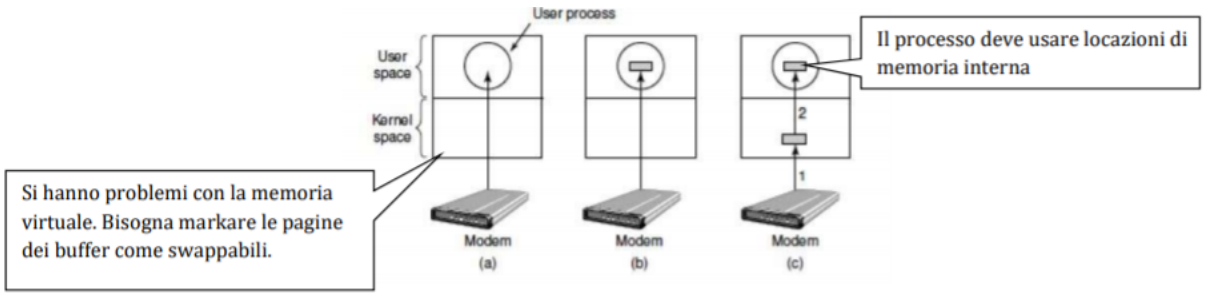
Buffering

Caching e Buffering sono diversi !

Il buffer è una zona di memoria usata temporaneamente per l'entrata o l'uscita dei dati, oppure per velocizzare l'esecuzione di alcune operazioni (ad esempio operazioni su stringhe). È una memorizzazione transitoria dei dati in memoria mentre essi sono trasferiti fra due dispositivi o tra un'applicazione ed un dispositivo.
Permette di disaccoppiare la chiamata di sistema di scrittura con l'effettiva uscita dei dati (output asincrono).

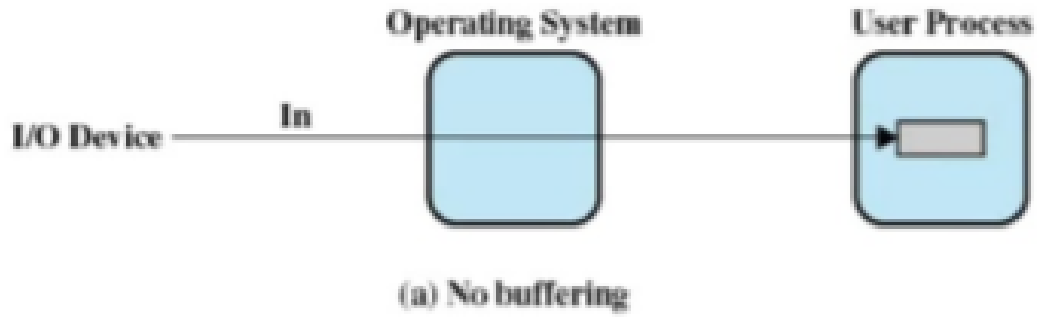
Il produttore ed il consumatore hanno velocità diverse (doppio buffer).
- I dispositivi trasferiscono dati in blocchi di dimensioni diverse.
L'uso eccessivo della bufferizzazione porta ad un peggioramento delle prestazioni.

Strategie di buffering:
- Non usare il buffer → inefficiente.
- Buffer nello spazio utente → problemi con la memoria virtuale.
- Buffer nel kernel → bisogna copiare i dati, con blocco dell'I/O nel frattempo.



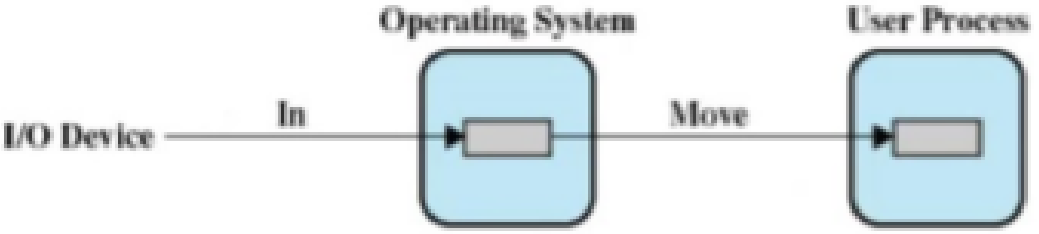
Possiamo avere diverse tipologie di buffer:

- **Senza buffer**: si ha inefficienza.



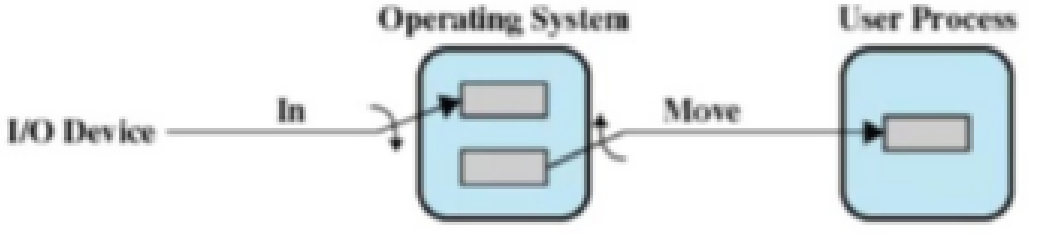
(a) No buffering

- **Buffer singolo**: se il produttore è più veloce del consumatore, può capitare che il produttore deve aspettare che il consumatore finisca e dunque il numero di content switch può aumentare.



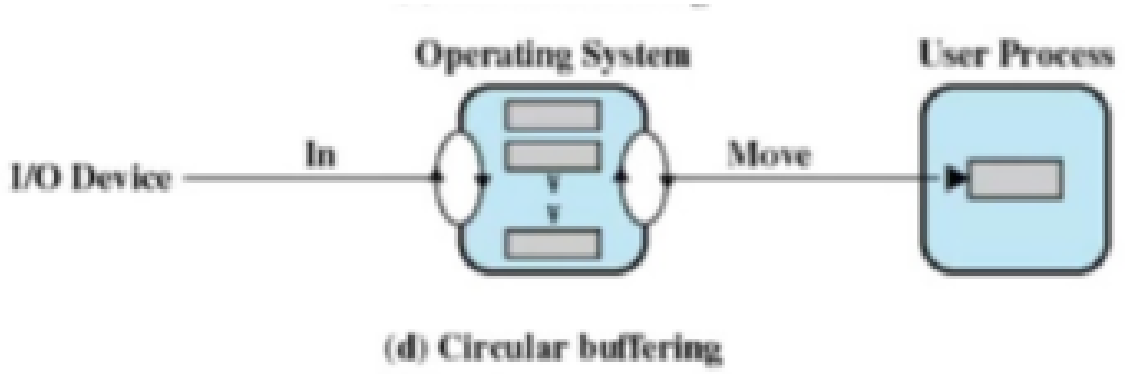
(b) Single buffering

- **Buffer doppio**: produttore e consumatore hanno due buffer separati. Evitiamo il problema del buffer singolo.



(c) Double buffering

- **Buffer circolare**: abbiamo più buffer, il produttore lavora su uno e il consumatore su un altro. Così facendo si riduce la probabilità che produttore e consumatore siano sullo stesso buffer.

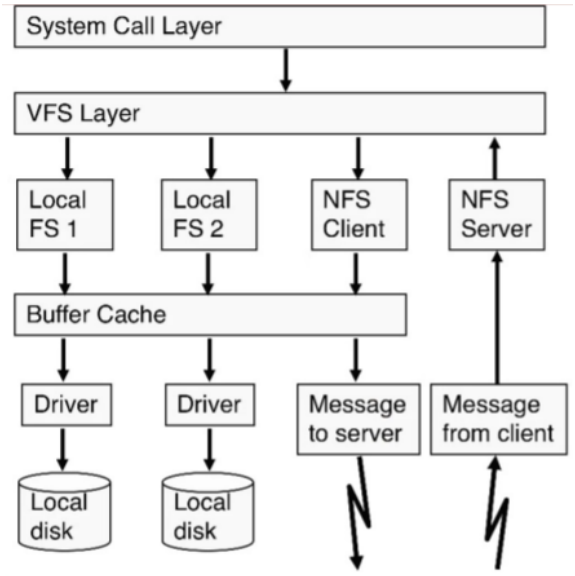


(d) Circular buffering

File System di Linux

Virtual File System

Adotta il file system virtuale.
Permette di lavorare con file system reali di natura differente nello stesso SO.
Ogni FCB, usando questo approccio, contiene anche il tipo di file system reale con cui viene salvato.



Ogni file ha i suoi metodi specifici del suo file system.

Nasconde i dettagli implementativi di ciascun tipo di file system e fornisce delle API comuni.
Il VFS è progettato seguendo il paradigma ad oggetti.

- inode-object, file-object → rappresentano file singoli.
- file-system-object → rappresenta un intero file system.

Layout di un file system:

Boot Block	Superblock	Gestione spazio libero	Gestione spazio occupato	Root dir	File e Directory

- Boot block: info per il bootstrap.
- Superblock: info sull'intero file system (tipo di file system, quanti blocchi esistono, dimensione dei blocchi, meta informazioni sul file system, informazioni specifiche del FS).
- Root dir: cartella di partenza.

Ext2 File System

Ext2fs utilizza un meccanismo di allocazione dei blocchi di un file simile a quello di BSD Fast File System (ffs).

Usa un meccanismo di allocazione che cerca di memorizzare blocchi logici contigui in blocchi fisici contigui, per quanto possibile, in modo da minimizzare gli accessi.

Suddivide il file system in gruppi di blocchi e tenta di assegnare ad un file solamente blocchi appartenenti allo stesso gruppo.
La frammentazione del file in blocchi diversi comporta un rallentamento nell'accesso del file. Ext2 cerca di evitare tale rallentamento allocando blocchi logicamente adiacenti in blocchi fisicamente adiacenti.

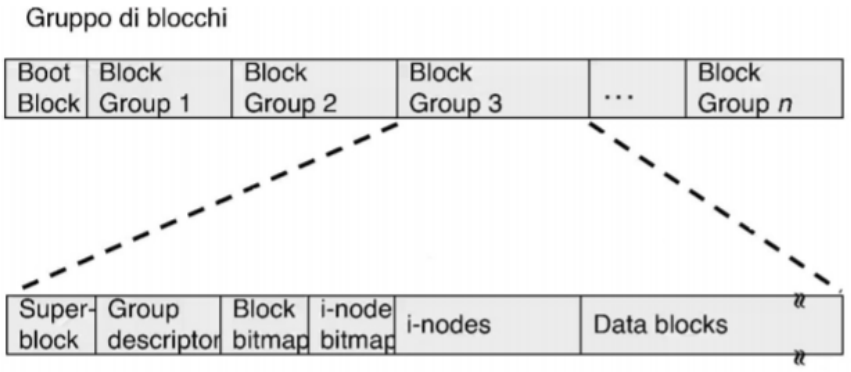
Una partizione è suddivisa in gruppi di blocchi.

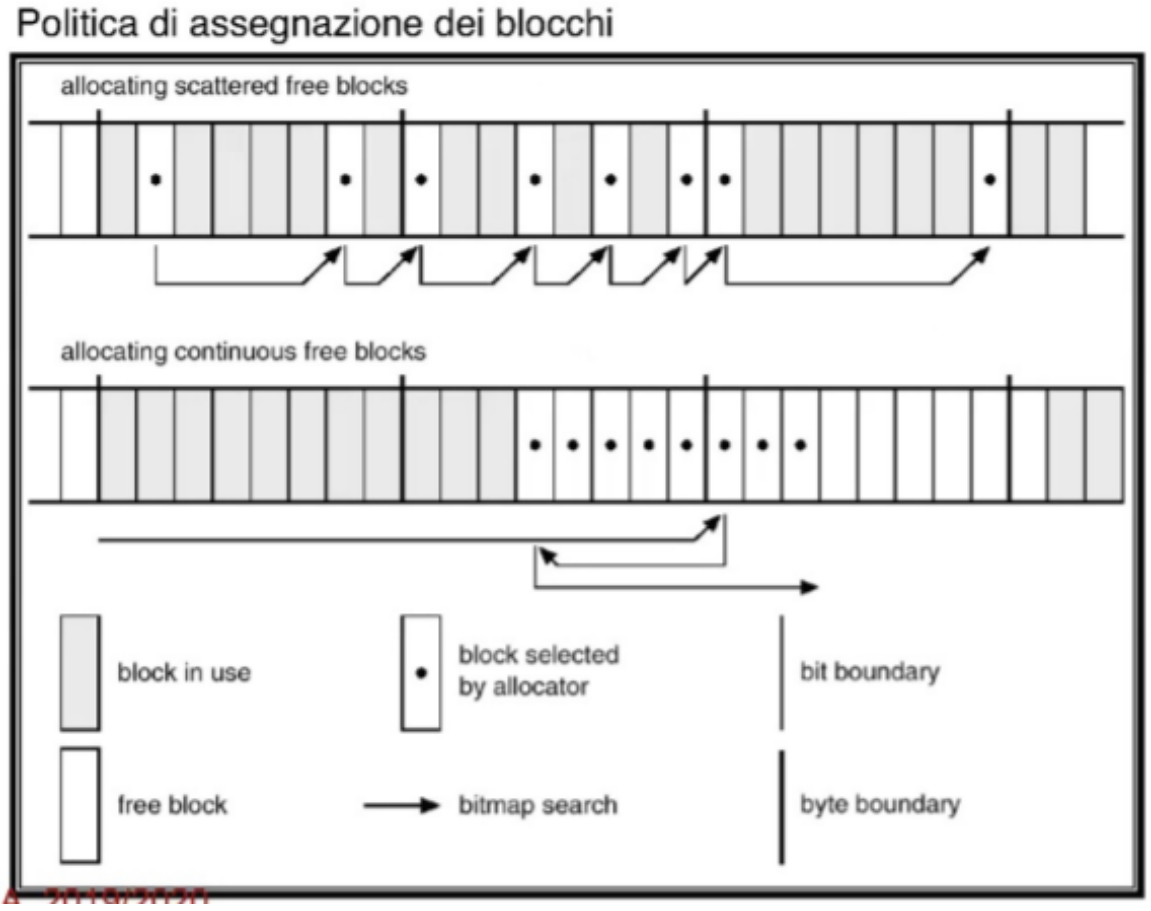
Si può riutilizzare la bitmap dei blocchi liberi poiché il gruppo di blocchi ha dimensioni adeguate per essere gestito dalla bitmap dei blocchi.

Le principali differenze sono:
- In ffs, i blocchi sono da 8 KB e sono suddivisi in frammenti da 1 KB per ridurre la frammentazione interna nell'ultimo blocco.
- Ext2fs non usa frammenti, esistono solo blocchi da 1 KB, anche se sono permessi blocchi da 2 KB e 4 KB.

- Ext2fs utilizza politiche di allocazione progettate in modo tale che una richiesta di I/O riguardante diversi blocchi può essere effettuata con una sola operazione:

- suddivide un file system in gruppi di blocchi e tenta di assegnare ad un file solo blocchi appartenenti ad un unico gruppo;
- tenta di posizionare blocchi logicamente adiacenti in blocchi fisicamente adiacenti sul disco.





Ext3/Ext4 File System

Sono versioni di journaling di Ext2 (ovvero con annotazione delle modifiche).
Ogni operazione viene fatta in 2 passi:
- una copia dei blocchi da scrivere viene scritta nel journal prima del commit;
- dopo il commit, i blocchi vengono scritti nelle effettive destinazioni.

In caso di mancanza improvvisa di alimentazione si possono ricostruire tutte le operazioni effettuate.

L'Ext3 presenta 3 modalità di funzionamento:
1)
Journal → rischio basso
Nel journal finiscono sia i metadati che i dati; molto costoso.
2)
Ordered → rischio medio
Prima vengono fatte le operazioni sui dati, poi i metadati vengono scritti sul journal, poi i metadati sono scritti nella destinazione effettiva.
3)
Writeback → rischio alto
Nel journal finiscono solo i metadati; più efficiente.

Proc File System

Il file system *proc* serve da interfaccia per altri servizi; non memorizza dati, il suo contenuto viene calcolato on demand.
proc deve implementare una struttura di directory e il contenuto dei file ; questo significa anche definire un i-node number univoco e persistente per ogni directory e file contenuto nel file system.
Esempio: il comando ps; ps gira in usermode.

Per vedere quali sono i processi dovrebbe accedere alla tabella dei processi, quindi dovrebbe essere lanciato in kernel mode. Per aggirare tale limitazione si utilizza l'interfaccia proc.
Proc è una routine del kernel organizzata come file system: leggendo la directory dei processi, ad esempio, partirà una funzione del kernel che legge dalla tabella dei processi. Proc permette di accedere alle strutture dati del kernel come se fossero dei file.

Il file system *proc* serve da interfaccia per altri servizi; non memorizza dati, il suo contenuto viene calcolato on demand.
proc deve implementare una struttura di directory e il contenuto dei file ; questo significa anche definire un i-node number univoco e persistente per ogni directory e file contenuto nel file system.
Esempio: il comando ps; ps gira in usermode.

Per vedere quali sono i processi dovrebbe accedere alla tabella dei processi, quindi dovrebbe essere lanciato in kernel mode. Per aggirare tale limitazione si utilizza l'interfaccia proc.
Proc è una routine del kernel organizzata come file system: leggendo la

directory dei processi, ad esempio, partirà una funzione del kernel che legge dalla tabella dei processi. Proc permette di accedere alle strutture dati del kernel come se fossero dei file.