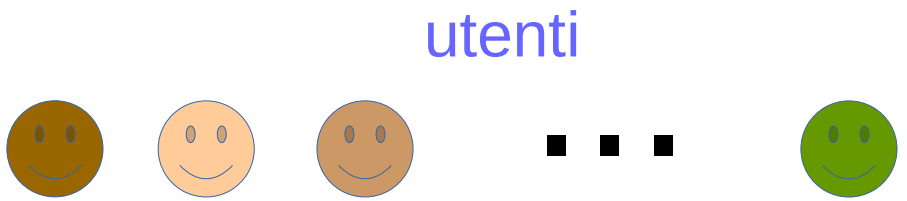
Implementazione del file system

Capitolo 11 del libro (VII ed.)

Introduzione

- Programmi e dati sono conservati in **memoria secondaria**
- la memoria secondaria ha le seguenti caratteristiche:
  - è **organizzata in blocchi** (un blocco può comprendere più settori)
  - è possibile **accedere direttamente** a qualsiasi blocco
  - è possibile leggere un blocco, modificarlo e riscriverlo esattamente **nella stessa posizione** di memoria
  - Si accede alla memoria secondaria attraverso un **FILE SYSTEM**

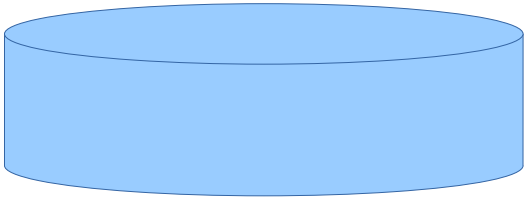
Il FS sta fra gli utenti e la memoria secondaria



fornisce una vista sulla memoria come organizzata in file e directory, definisce i metadati, fornisce operazioni permesse su file e directory

File System

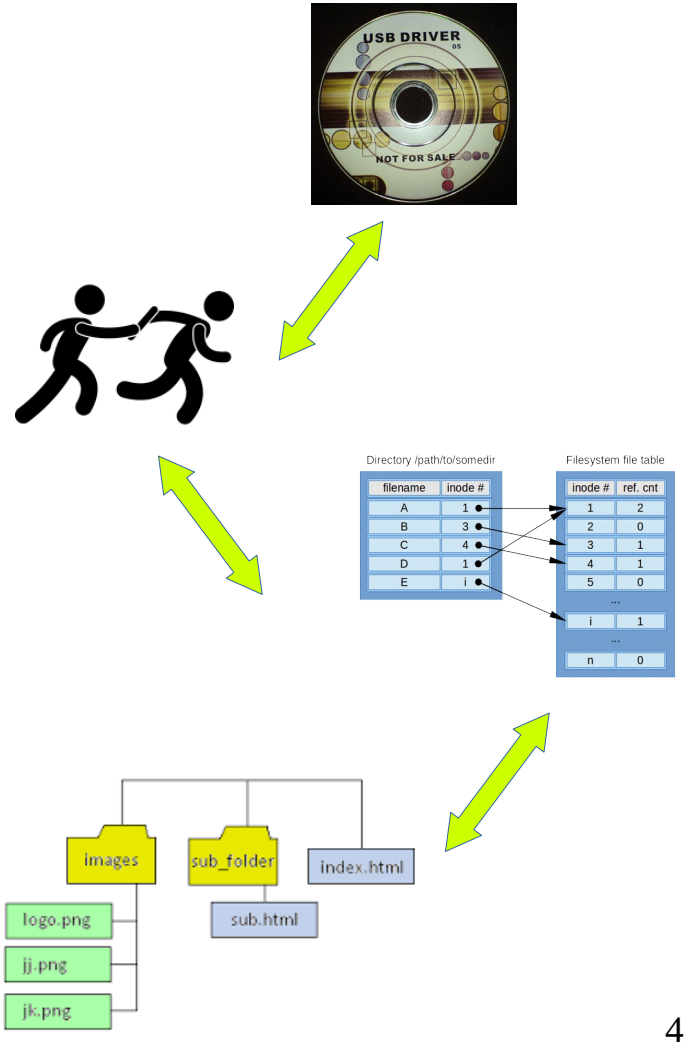
algoritmi e strutture dati che fanno corrispondere tutti gli elementi del FS visto dall'utente (FS logico) ai supporti di memoria su cui i dati sono conservati



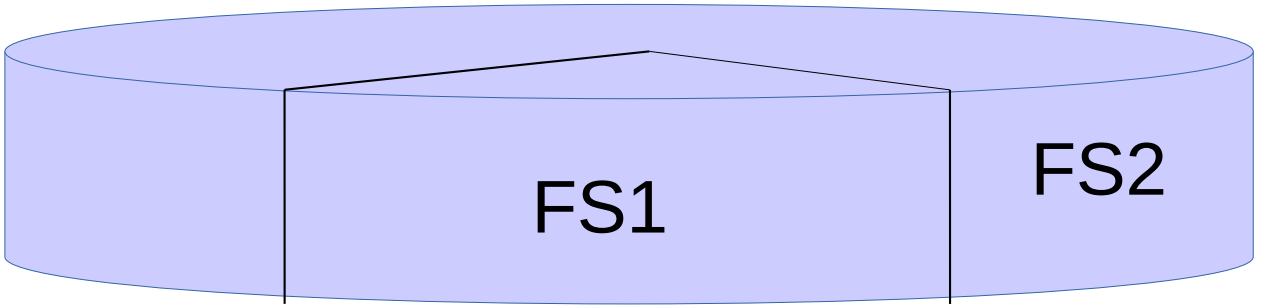
Memoria secondaria

Livelli del file system

- il FS di solito è strutturato in una sequenza di livelli, dal basso verso l'alto:
  - **driver di dispositivo**: si occupa del trasferimento dei dati da dispositivo di memoria secondaria a RAM e viceversa. È il gestore del dispositivo
  - **file system di base**: è proposto a passare comandi al driver di dispositivo
  - **modulo di organizzazione dei file**: è a conoscenza di come i file sono memorizzati su disco, è in grado di tradurre indirizzi logici in indirizzi fisici. Mantiene e gestisce anche l'informazione relativa ai blocchi liberi
  - **file system logico**: gestisce il FS a livello di metadati. Ogni file è rappresentato da un **File Control Block** (FCB)



Struttura del disco



- Un disco può essere diviso in più **partizioni**
- Ogni partizione è organizzata in **un solo FS**
- Un FS contiene il SO e/o i file degli utenti
- I diversi FS sono composti in una sola struttura (operazione **mount**)

Struttura su disco

- Un FS è una sequenza di blocchi di memoria secondaria
- Blocchi speciali:
  - **boot control block**: se il FS non contiene un SO è un blocco vuoto, se invece contiene un SO, questo blocco contiene info necessarie in fase di bootstrap

Struttura su disco

- Blocchi speciali:
  - boot control block: ...
  - **volume control block** (o **superblocco**): descrive lo stato del FS stesso, quant'è grande, quanti file può contenere, ecc. ... In particolare, contiene il numero dei blocchi appartenenti alla partizione, la loro dimensione, il numero di blocchi liberi (e loro riferimenti)

Struttura su disco

- Blocchi speciali:
  - boot control block: ...
  - volume control block (o superblocco): ...
  - **struttura delle directory**: organizza i file, è implementata in modi diversi e contiene coppie <nome file, FCB>

Struttura su disco

- Blocchi speciali:
  - boot control block: ...
  - volume control block (o superblocco): ...
  - struttura delle directory: ...
  - una **lista di FCB** (file control block): contengono i metadati relativi ai file e consentono di accedere ai loro contenuti;gli FCB sono anche detti **inode** (index-node), ciascuno identificato da un numero

INODE / FCB

- Un **FCB** (o **inode**) mantiene le informazioni relative ai file
- Gli inode sono conservati in **memoria secondaria**
- **Esempio**: in Unix un inode su disco può contenere queste informazioni

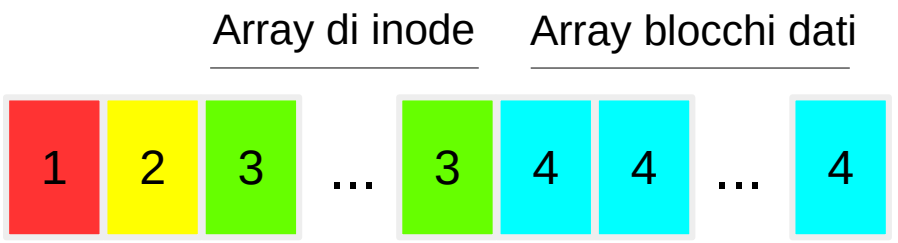
• Identità del proprietario del file	(User ID)
• Tipo del file	(regolare, directory, link, device, ...)
• Diritti di accesso	(r w x per proprietario, gruppo, altri)
• Tempi di accesso e modifica	(data/ora ultimo accesso/ultima modifica) # di link al file
• Dimensione del file numero di byte	
• Tabella per l'accesso ai dati	(indirizzi dei blocchi di mem. secondaria contenenti i dati)

NB: per semplificare l'accesso gli inode hanno **dimensione fissa** -> es. è prevista una lunghezza massima per i nomi dei file, l'indirizzamento della memoria avviene attraverso parole di lunghezze prefissata

Esempio: FS del SO MINIX

- Minix è un SO scritto in C da Andrew Tanenbaum (Vrjie University) per scopi didattici (fine anni '80)
- I nomi di file sono al più di 14 caratteri
- i blocchi possono essere riferiti da parole di 16 bit (2^{16} blocchi) - questo impone un limite importante, se i blocchi sono grandi 1KB non è possibile gestire memorie secondarie più grandi di 64 MB
- Linux è nato come tentativo di superare questi due limiti di Minix

FS di MINIX



1 Boot block: contiene un codice di bootstrap usato per avviare il SO

2 Super block: descrive lo stato del FS (quant'è grande, quanti file può contenere, ecc.)

3 Lista degli inode: ogni inode corrisponde a un file (esistente o potenziale), uno in particolare corrisponde alla radice del FS

4 Blocchi dati: ogni blocco può appartenere a uno e un solo file

NB: in Unix le **directory** sono implementate come **file speciali,** quindi alcuni inode corrispondono a directory

Strutture dati in RAM

- All'avvio di un elaboratore viene “montato il FS” e vengono caricate in RAM delle informazioni che riguardano la sua struttura
- Lo scopo è consentire l'accesso (efficiente) ai file
- Le strutture mantenute in RAM sono:
  - **mount table**: mantiene informazioni su ciascuna partizione montata, ogni operazione di mount/unmount la modifica
  - **directory**: mantiene informazioni sulle directory a cui si è avuto accesso di recente (utile in particolare per quei SO che usano descrittori diversi per i file e per le directory. Alcuni SO, e.g. Unix, invece implementano le directory come file speciali)
  - **tabella dei file aperti**: mantiene una copia arricchita dei FCB di tutti i file aperti al momento tabelle dei file dei processi: si ha una tabella di questo tipo per ogni processo; per ogni file aperto viene mantenuto un riferimento all'opportuno elemento della tabella (globale) dei file aperti

In-core inode (FCB in RAM)

Quando un file viene aperto per la prima volta il suo inode viene caricato in RAM ed arricchito da qualche informazione aggiuntiva

Esempio: in Unix un inode su ram può contenere queste informazioni

- **INODE COPIATO DA DISCO**
- **Stato dell'in-core inode**: dice se
  - l'inode è locked (il file non è disponibile)
  - la copia dell'inode in RAM è diversa da quella su disco,
  - il file è un punto di mount
  - ...
- **device number**: identificatore del FS a cui appartiene il file
- **inode number**: identificatore dell'inode nella struttura dati su disco
- **contatore** del numero di riferimenti all'inode (#utilizzi del file attuali)
- ...

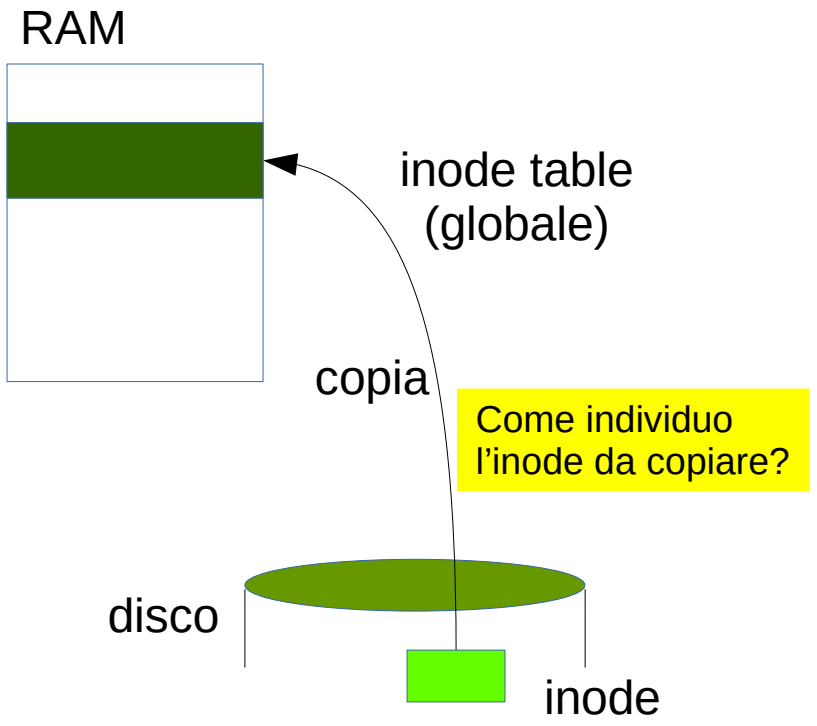
In Unix gli node caricati sono detti **incore inode** e sono conservati in una tabella globale che si chiama **inode table**

Apertura di un file

- supponiamo che un processo esegua la sys. call **open("prog.c", O_RDONLY)**

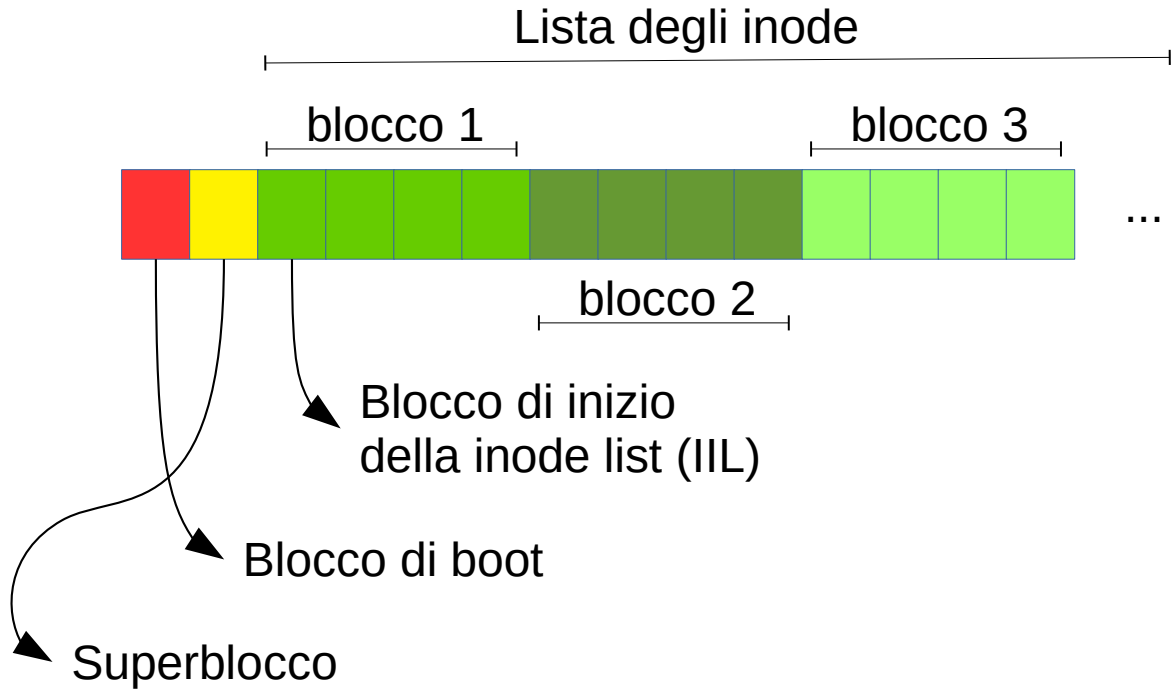
- Il kernel mantiene una **inode table** di dimensione finita
- Nell'eseguire la open: controlla se l'inode corrispondente al file è già stato caricato, **se sì userà l'incore inode trovato** altrimenti:
- Se c'è spazio, crea un **nuovo incore inode**, lo "locka" e va a copiarvi l'inode su disco corrispondente al file da usare
- Se non c'è spazio l'operazione fallisce e viene restituito un errore (così il processo richiedente non rischia di rimanere sospeso per tempi lunghi)
- *se esisteva già .... lo vedremo fra poco*

FAIL



- **Nota**: il lock (esclusivo e obbligatorio) serve per evitare inconsistenze. Si attiva all'inizio di certe system call e si disattiva al loro termine

Esempio: UNIX



La lista degli inode è contenuta in una **sequenza di blocchi**

Gli inode hanno dimensione fissa

Tutti i blocchi contengono lo stesso numero di inode
Esempio #inode-per-blocco: 4

Per accedere alla porzione di disco che memorizza un inode basta conoscerne: 1) il numero, 2) la dimensione degli inode (la stessa per tutti) e 3) quella dei blocchi

Il SO ha tutte queste informazioni

Proviamo a identificare l'indirizzo di inizio dell'inode con indice 5 (partendo a contare dall'indice 0)

Esempio: UNIX

L'inode è identificato da un indirizzo <B, D>:

B = blocco
S = displacement nel blocco

[1] Individuare il blocco che contiene l'inode
B = (inode-number / #inode-per-blocco) + IIL
/ : divisione intera

[2] Individuare il displacement dell'inode nel blocco
D = (inode-number % #inode-per-blocco) X dim-disk-inode
% : resto della divisione intera

- #inode-per-blocco: 4
- inode-number: 5
- B: IIL + 1
- D: 1 x dim-inode

Algoritmo NAMEI

- Nei lucidi precedenti abbiamo implicitamente supposto di avere a disposizione il numero dell'inode corrispondente al file di interesse
- Di solito però avremo il **nome del file** … o più in un cammino, es.

  mieiFile/Documenti/anno2020/slideSisOp.odp
- Per identificare l'inode relativo a un file devo avere a disposizione un algoritmo che è in grado di percorrere il cammino fino a identificare il numero dell'inode:

1) accedo alla directory *mieiFile*
2) cerco fra i suoi contenuti *Documenti*
3) accedo alla directory *Documenti*
4) cerco fra i suoi contenuti *anno2020*
5) accedo alla directory *anno2020*
6) cerco fra i suoi contenuti *slideSisOp.odp*
7) trovando il numero di un inode

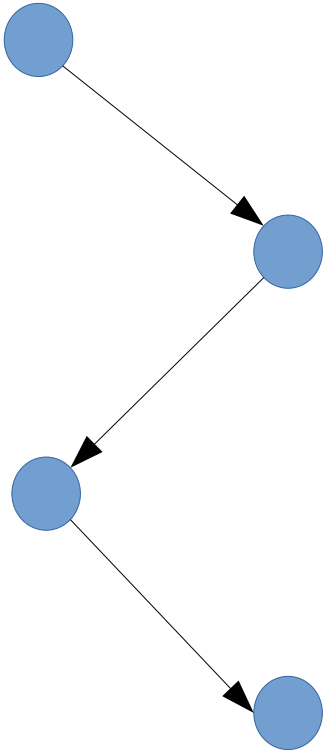
Algoritmo NAMEI

inode-number NAMEI(string cammino)

if (la prima directory del cammino è “/”)
    current = root inode
else current = inode della working directory

repeat
  el = leggi prossimo elemento dall'input;
  if (el is null) return current;
  else if (el è contenuto in current)
                    current = inode associato a el
        else return (no inode)
until (el is null);

return current;



Per verificare se el è contenuto in current occorre accedere ai blocchi dati relativi a current

Open di un file in Unix

- Un processo esegue **open(pathname, flags)**:
  - Il SO verifica se il file è in uso da parte di qualche processo;
  - **Se no**:
    (1) Utilizza l'algoritmo **namei** per identificare il numero di inode del file
    (2) Calcola l'indirizzo su disco **<B, D>**, che permette di accedere **all'inode** conservato su disco
    (3) Invia al **controller** del disco un comando di lettura che si concluderà con la copiatura dell'inode in una entry della tabella degli inode in RAM (**in-core inode**)
    (4) Aggiorna le **tabelle di sistema** di conseguenza
  - **Se sì**:
    (1) Identifica l'in-core inode e lo rende accessibile al processo che ha eseguito open modificando le **tabelle di sistema** contenute in RAM

Open di un file (UNIX)

La system call open(“prog.c”, O_RDONLY) restituisce come handle del file un **file descriptor**, cioè un numero intero. Cosa rappresenta questo numero?

In Unix vengono mantenute in RAM 3 tabelle:

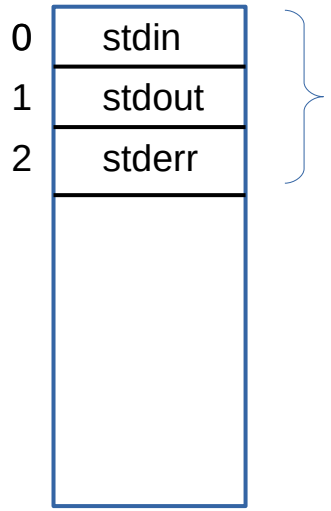
1) **tabella degli incore inode** **(globale)**: contiene gli inode dei file aperti

2) **tabella dei file** **(globale)**: contiene un riferimento alla tabella precedente, un contatore e un puntatore alla locazione del file a cui si è arrivati a leggere (o in cui scrivere)

3) tabella dei file del processo **(locale, 1 tabella per ogni processo)**: contiene riferimenti alla tabella (globale) dei file, uno per ogni file aperto, più 3 riferimenti presenti per ogni processo, anche se non ha file aperti:
   (a) *standard input*
   (b) *standard output*
   (c) *standard error*

Tabella dei file del processo

- Alla creazione di un processo viene creata la **tabella dei file aperti** specifica del processo
- Tale tabella contiene 3 entry corrispondenti a stdin, stdout, stderr:



Flussi standard che permettono l'interazione del processo con il suo ambiente (default: lettura da tastiera, scrittura su terminale, scrittura su terminale)

Tabella dei File del processo

Open

la system call **open(...)** restituisce come **handle del file** un **file descriptor**, cioè un numero intero

Per ogni open:

(1) Se il file era già stato aperto si aggiorna la entry corrispondente nella **tabella degli incore inode** (globale), altrimenti si crea una nuova entry (NB: se non c'è spazio: interrupt);

(2) Viene creata una entry nella **tabella dei file** (globale):

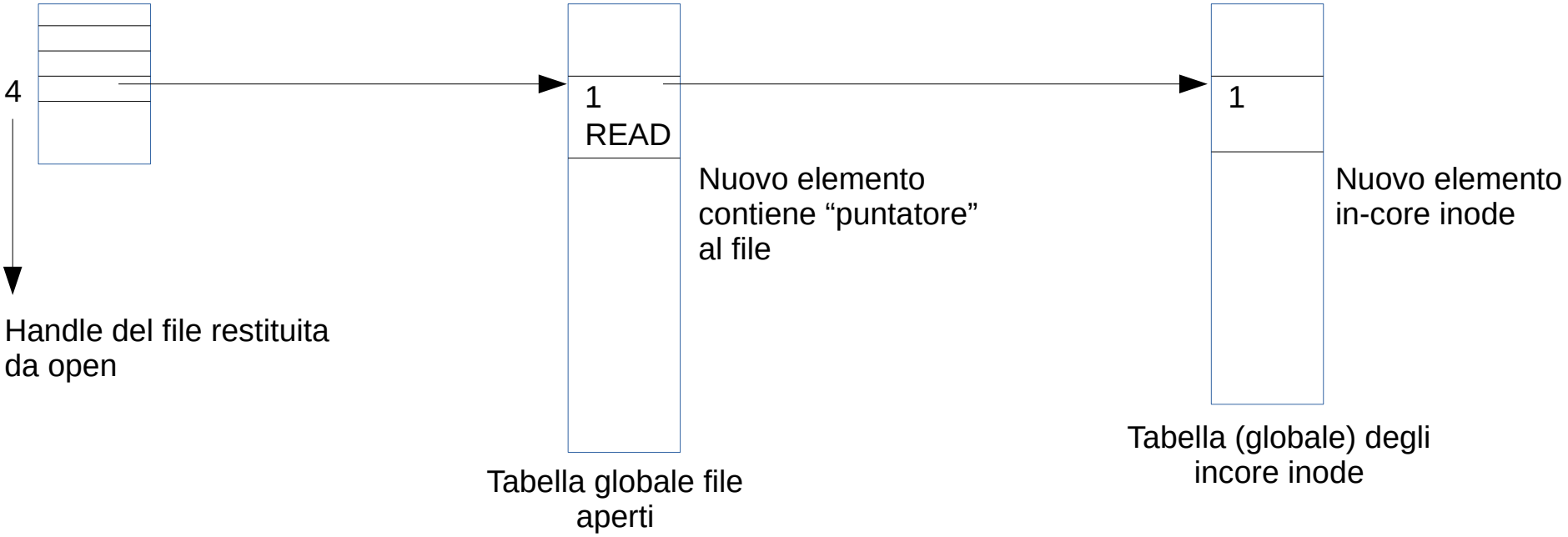
conterrà un riferimento all'inode più un displacement che indica l'indirizzo della prossima lettura (o scrittura). Il default per il displacement è zero (file aperto nel suo punto di inizio); è invece uguale alla dimensione del file se l'apertura avviene in modalità write/append

(3) Viene aggiunta una entry alla **tabella dei file** (privata del processo) che esegue la open. Questa entry contiene un riferimento alla entry nella tabella dei file globale.

NB: il suo indice in tabella è il file descriptor restituito da open

Il processo esegue una open

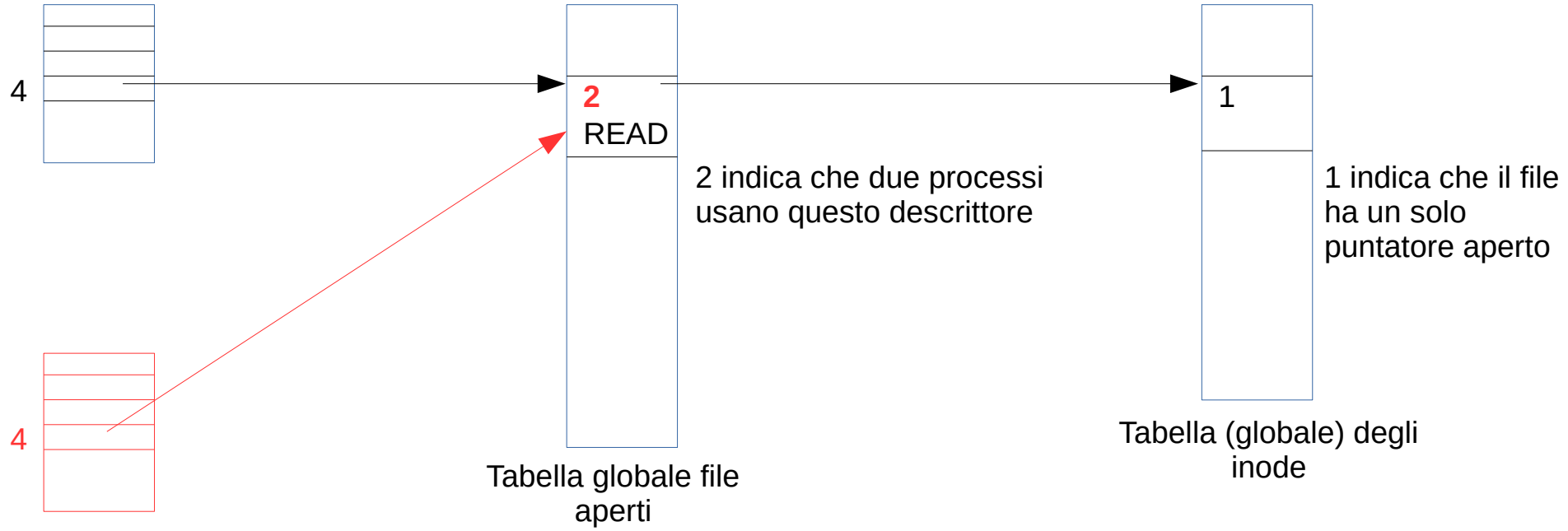
- Il processo esegue **open(O_RDONLY,"prova")**. Supponiamo che il file **non** sia in uso da parte di altri processi: il suo inode deve essere caricato in RAM



Ricorda: queste tabelle hanno una capienza limitata! Potrebbe non esseri spazio per un nuovo descrittore!

Se un processo figlio è creato

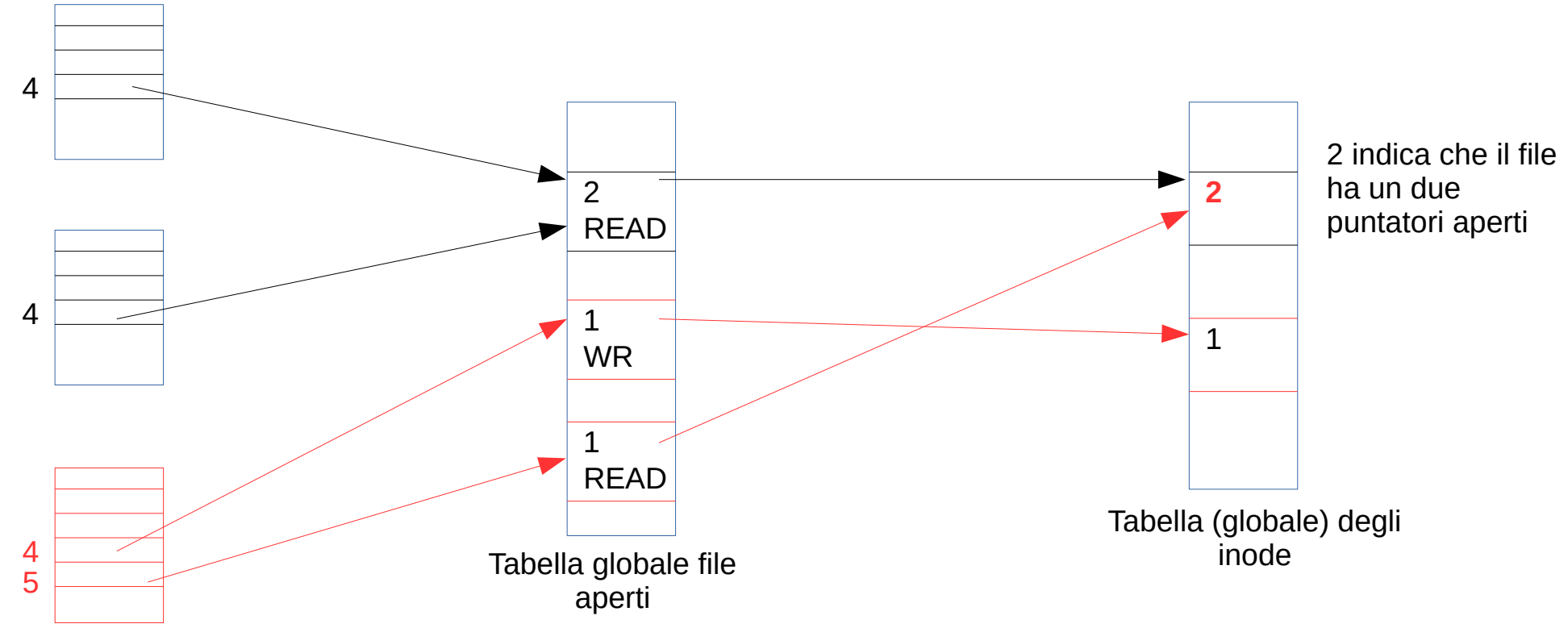
- Il processo esegue fork: il figlio eredita i descrittori dei file aperti contenuti nella tabella globale dei file aperti perché la sua tabella locale dei file è una copia di quella del padre



Handle del file restituita da open

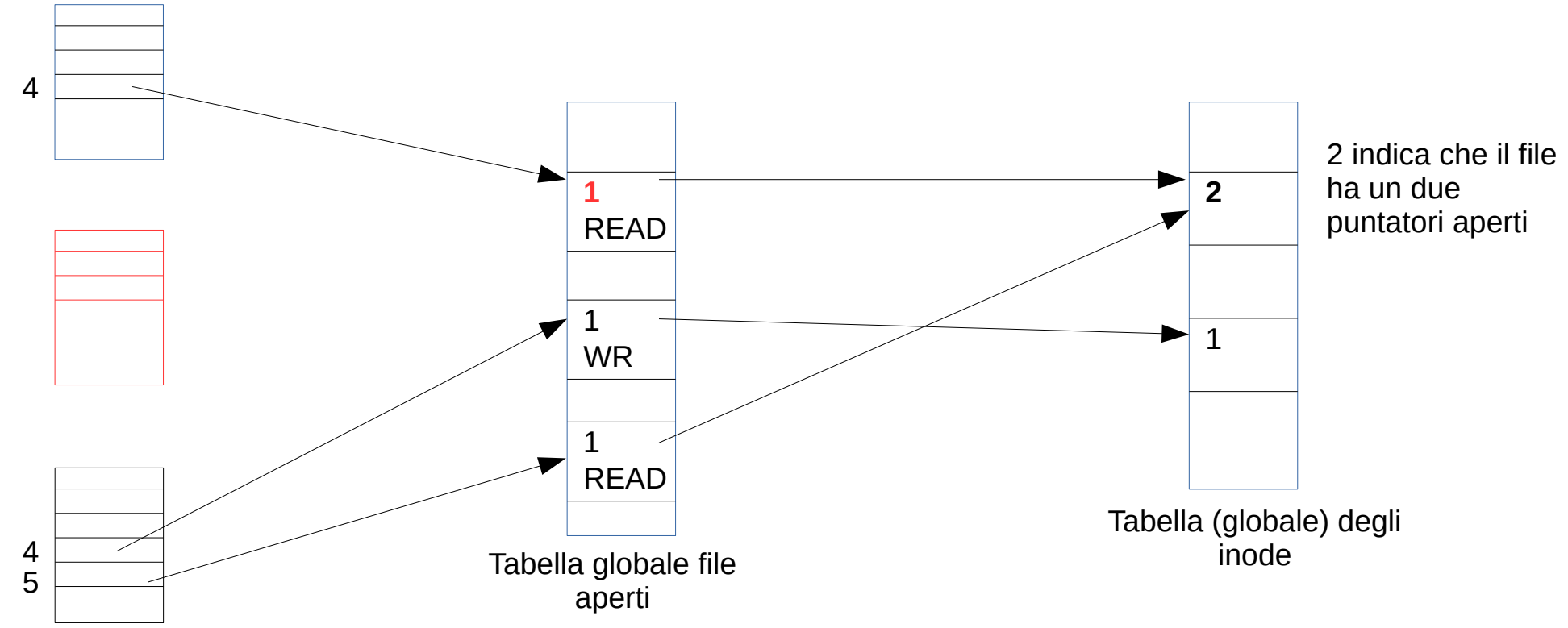
Un altro processo esegue open

- Un processo non in relazione di parentela con i precedenti esegue open(O_RDONLY, “prova”), dopo averne aperto un altro



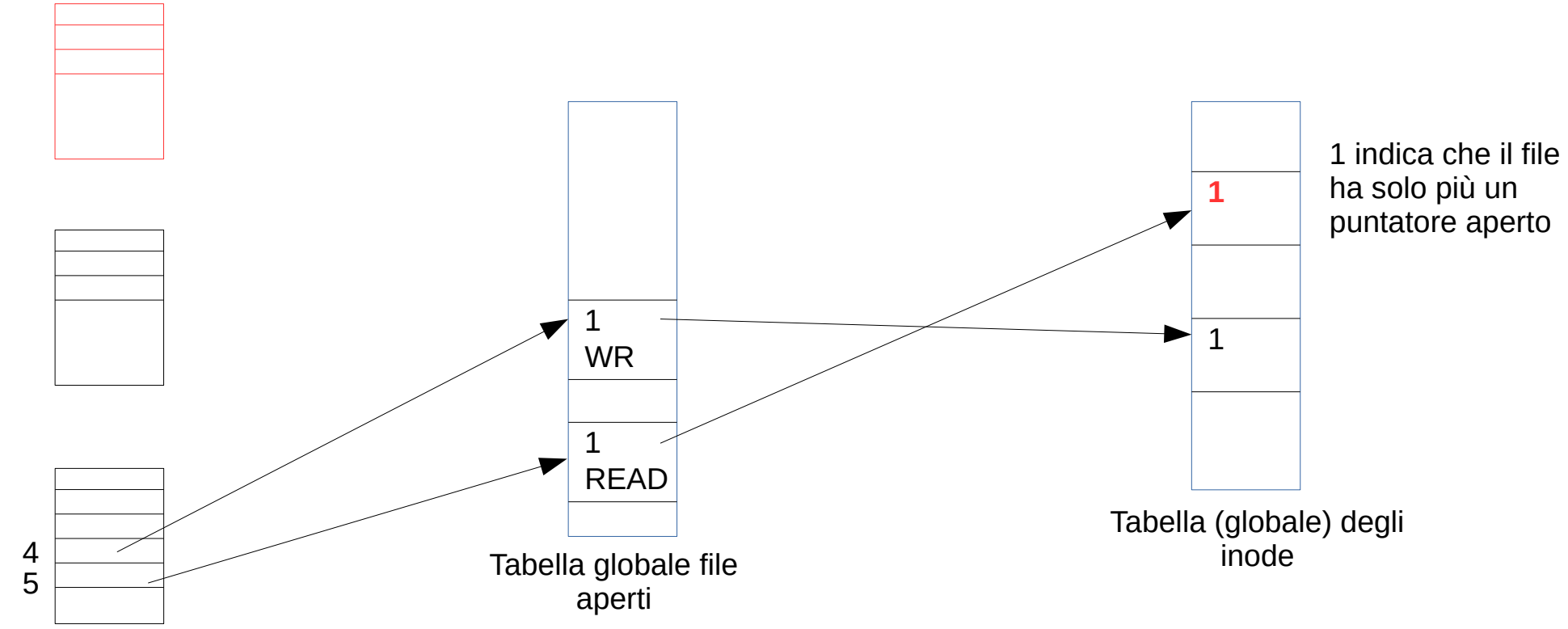
Il processo figlio esegue close

- Il processo figlio esegue close(4). Il puntatore in letture NON viene rimosso perché è ancora usato dal processo padre



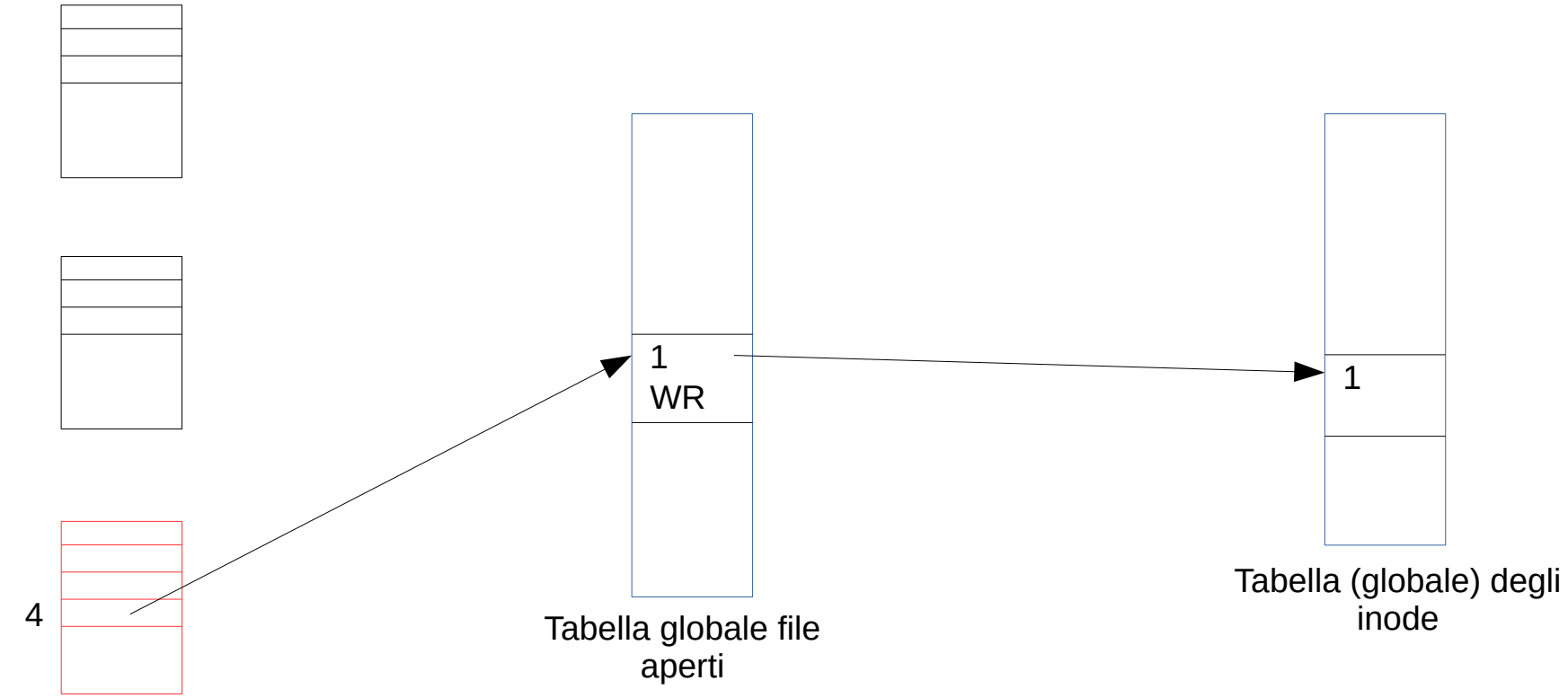
Il processo padre esegue close

- Il processo padre esegue close(4). Il puntatore in letture VIENE rimosso perché serve più. Il numero di riferimenti all'inode è decrementato



Il terzo processo esegue close

- Il terzo processo esegue close(5). Anche l'inode è rimosso perché nessun processo usa più il file. I frame di RAM occupati dai suoi dati sono liberati.



Implementazione delle directory

- La scelta di come implementare le directory è un punto critico nella progettazione di un File System
- Fra le varie tecniche:
  - **lista lineare (alla Unix)**: una directory è una sequenza di coppie <nome_file, inode number>
  - **B-tree**
  - **tabella hash**

Sequenza lineare

Di facile realizzazione

Limiti

- per verificare se un file è contenuto nella directory occorre scorrere il contenuto della directory
- ricerca di tipo lineare (lenta)
- è difficile mantenere ordinata la lista senza appesantire la gestione delle directory
- occorre avvalersi di strutture d'appoggio e implementare particolari algoritmi per gestire i buchi che si creano nelle directory con la
- cancellazione di file

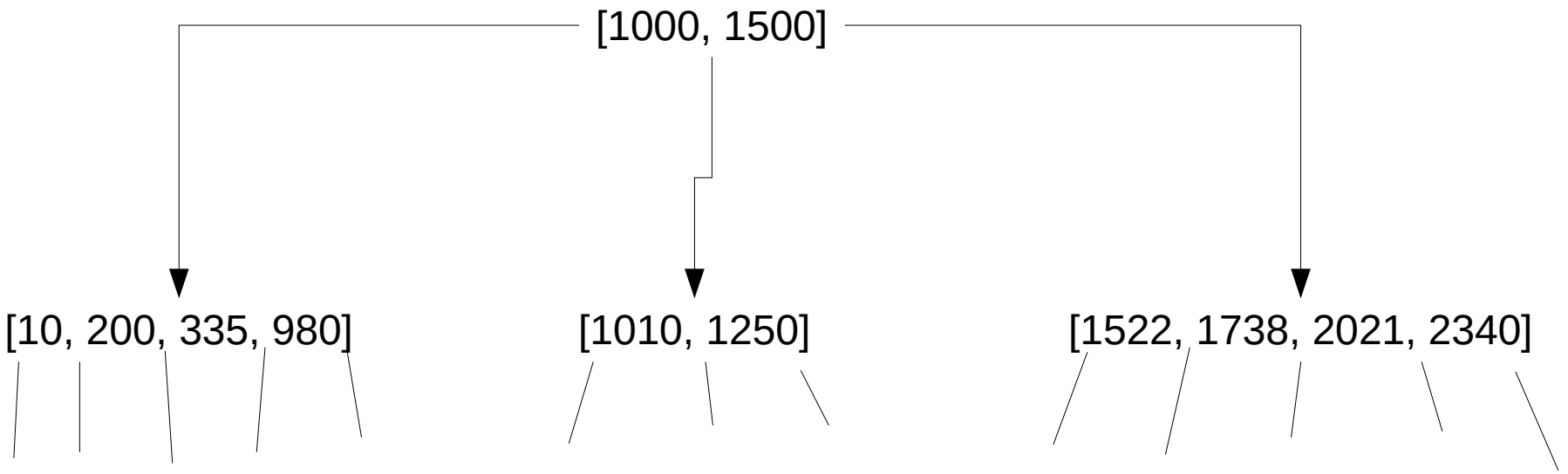
Possibili migliorie?

- appoggiarsi a strutture non lineari, in particolare alberi

B-Tree

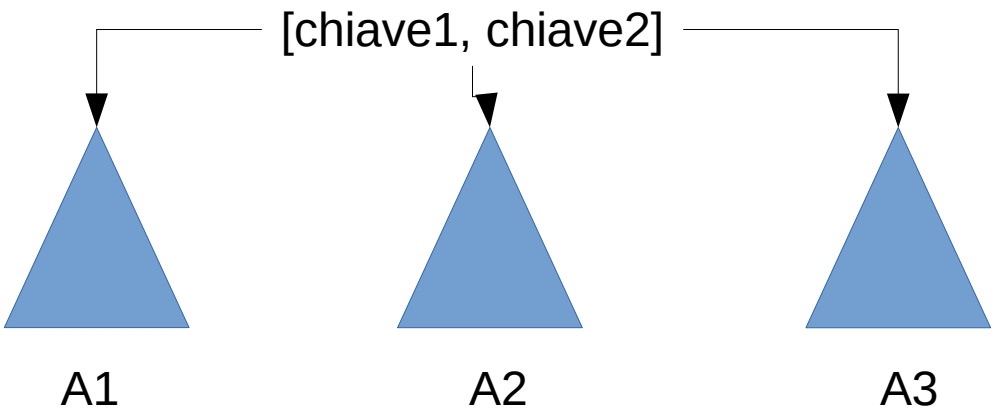
- un B-tree è una struttura ad albero ordinato.
- Ogni nodo contiene da **N** a **2N** elementi detti **chiavi**.
- Se un nodo contiene **K** elementi, allora avrà **K+1 puntatori** a nodi del livello successivo.
- Il numero N è detto **ordine dell'albero.** Per esempio il successivo è un albero di ordine 2 ...

B-Tree di ordine 2: esempio



Se i numeri fossero identificatori di inode con associati i relativi metadati questo albero potrebbe rappresentare i contenuti di una directory

In generale



Tutte le chiavi del sottoalbero A1 sono minori di chiave1

Tutte le chiavi del sottoalbero A2 sono comprese fra chiave1 e chiave2

Tutte le chiavi di A3 sono maggiori di chiave2

In un FS le chiavi possono essere identificatori di inode; ad ogni identificatore è associato anche l'inode relativo

B-tree

- La ricerca comincia sempre dalla radice del B-tree
- l'ordinamento dell'albero consente di trovare qualsiasi elemento in tempi rapidissimi per es.:

  in un B-tree di ordine 25 con 10.000.000 di nodi possiamo individuare qualsiasi chiave attraversando al più 4 nodi, se l'albero è bilanciato

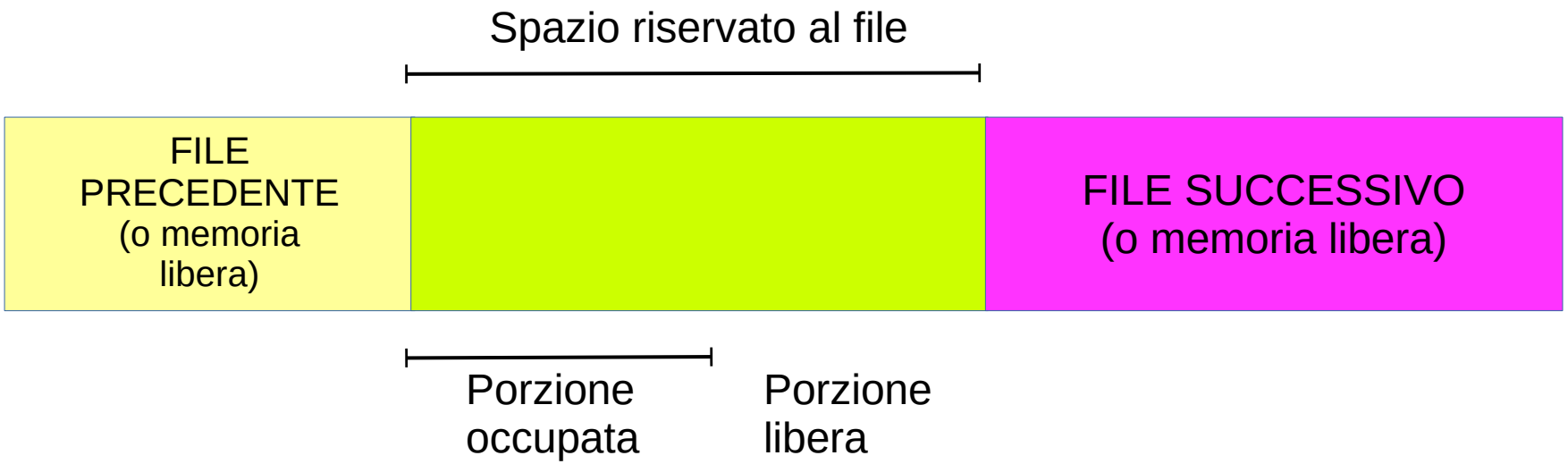
- Bilanciamento (#nodi sottoalberi sinistri = #nodi sottoalberi destri) e fan-out (apertura dell'albero) sono due caratteristiche fondamentali delle strutture ad albero usate per la ricerca

Allocazione dello spazio disco ai file

- Com’è organizzata la memorizzazione dei dati su disco?
  - allocazione contigua
  - allocazione concatenata (variante: FAT)
  - allocazione indicizzata

Allocazione CONTIGUA

Ogni file è allocato in una **sequenza contigua di blocchi**



Allocazione CONTIGUA

Ogni file è allocato in una **sequenza contigua di blocchi**

Vantaggi

- **rapidità di accesso al file**: il tempo di seek (posizionamenti della testina sulla traccia giusta) è trascurabile.
- Quando si accede a un file si tiene traccia dell'ultimo blocco letto, quindi se abbiamo appena letto il blocco B, sia l'**accesso sequenziale** (al blocco B+1) sia l'**accesso diretto** (al blocco B+k) sono immediati

Allocazione CONTIGUA

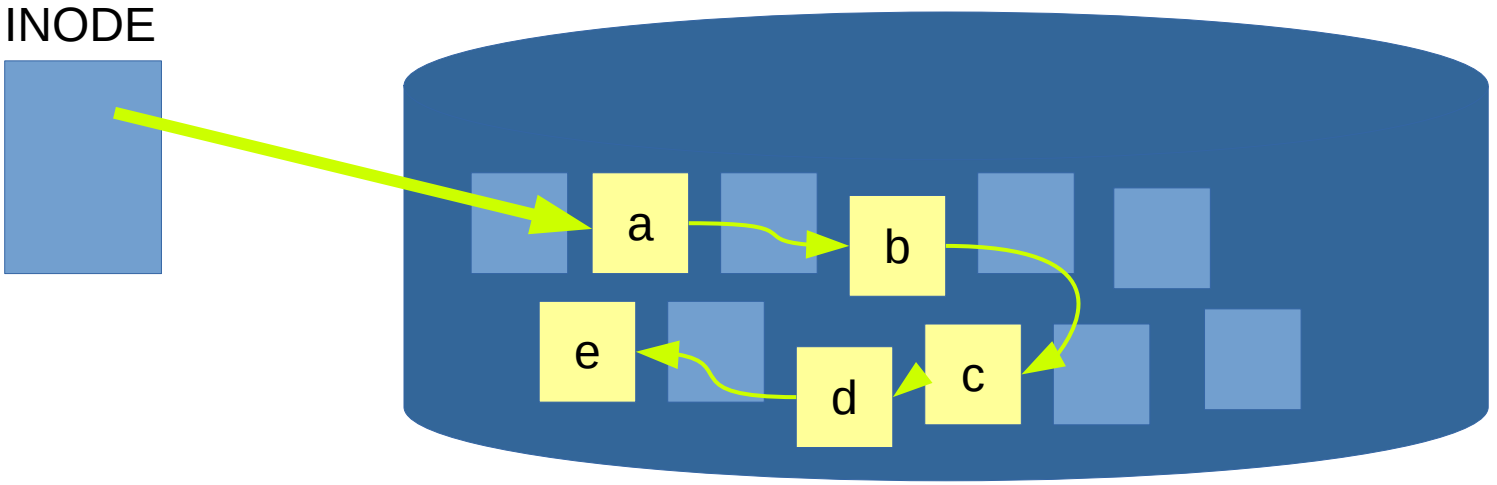
Ogni file è allocato in una **sequenza contigua di blocchi**

Svantaggi

- quanta memoria riservo per un file?
- se ne riservo troppa e il file cresce lentamente spreco memoria
- se ne riservo troppo poca? Necessità di estensioni!
- gestione dei buchi di memoria libera (best-, first-, worst-fit)
- frammentazione esterna
- necessità di deframmentare il disco di tanto in tanto

Allocazione concatenata

- Alternativa: spezzare il file in parti che possono essere allocate in modo non contiguo (blocchi dati). Occorre della sovrastruttura per mantenere l'informazione che certi blocchi dati sparsi sul disco costituiscono le parti di uno stesso file
- L'allocazione concatenata consente di fare proprio questo un file è costituito da una sequenza di blocchi sparsi per il disco ma mantenuti in una lista concatenata



Allocazione concatenata

Vantaggi

- non è necessario preallocare memoria per i file
- la lista concatenata è dinamica per natura
- non è necessario effettuare alcuna deframmentazione

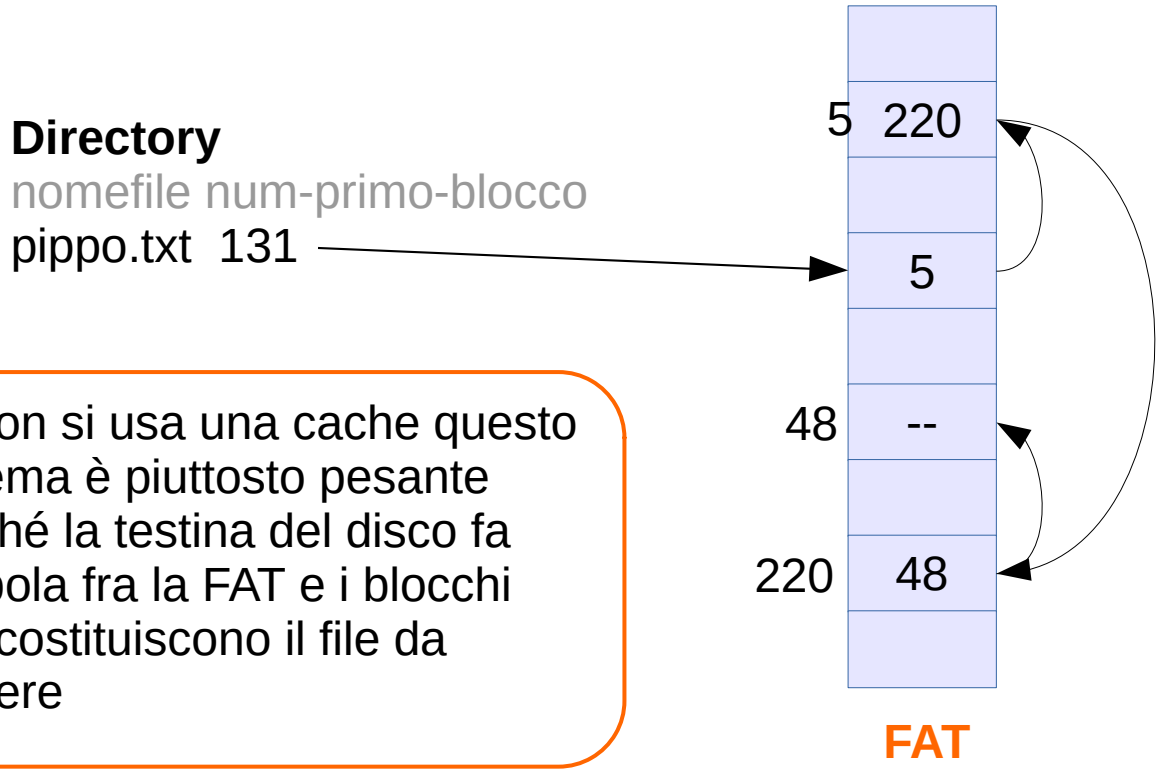
Allocazione concatenata

Svantaggi

- solo l'accesso sequenziale è efficiente, accesso diretto e indicizzato richiedono di scorrere la lista dei blocchi comunque
- i puntatori ai blocchi sono sparsi per il disco ogni salto di blocco comporta un tempo di latenza (tempo sprecato)
- occorre spazio per mantenere i puntatori ai blocchi successivi
- se un puntatore si corrompe si perde tutta la porzione successiva di file (!!!)

Allocazione concatenata: File Allocation Table (FAT)

- usata nei sistemi MS-DOS e successori
- si riserva una sezione della partizione per mantenere una tabella che ha tanti elementi quanti blocchi. Se un blocco fa parte di un file, il contenuto della entry corrispondente in tabella è un riferimento al blocco successivo:



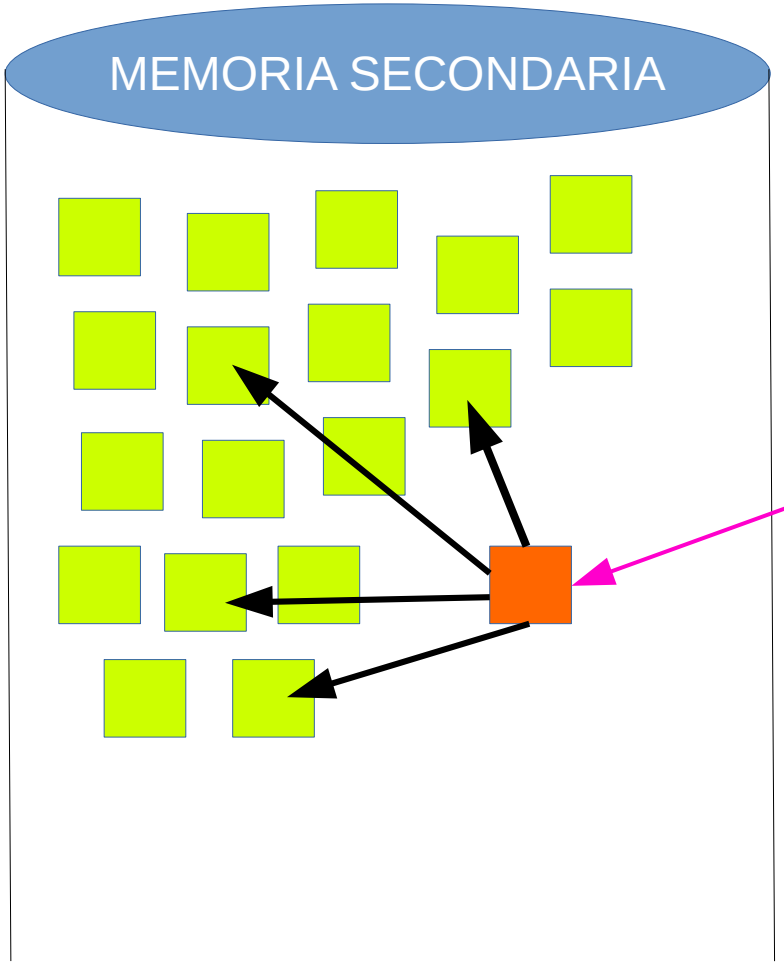
se non si usa una cache questo schema è piuttosto pesante perché la testina del disco fa la spola fra la FAT e i blocchi che costituiscono il file da leggere

Una FAT mantiene la struttura a lista concatenata esternamente ai blocchi dei file veri e propri

Allocazione indicizzata

- Tipo di allocazione che tenta di superare i limiti delle soluzioni precedenti introducendo un blocco indice
- Ogni file ha un **blocco indice**: **un array contenente gli indirizzi dei blocchi che costituiscono il file**
- I riferimenti ai blocchi indice dei file sono mantenuti nelle **directory**
- **Accesso**:
  - tramite la directory recupero il blocco indice
  - tramite i riferimenti contenuti nel blocco indice posso accedere ai vari blocchi dati

Blocco indice



Directory

File1 bloccoindice1
File2 bloccoindice2
...

Quando il file viene creato, tutti gli elementi del suo blocco indice sono NULL

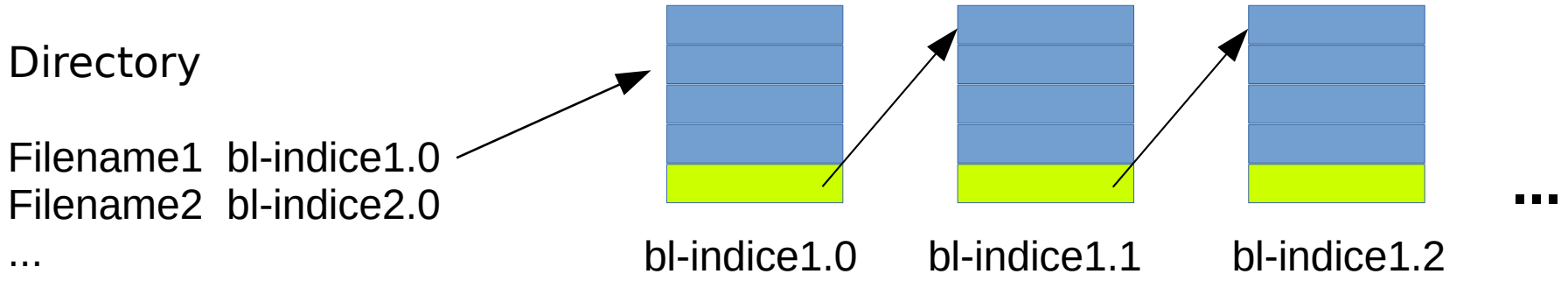
Man mano che il file cresce di dimensioni vengono allocati nuovi blocchi e i loro riferimenti sono inseriti in ordine nel blocco indice

Blocco indice

- La soluzione a blocco indice **elimina il problema della frammentazione esterna** in modo del tutto analogo all'organizzazione della RAM in pagine di pari dimensione, allocate quando c'è bisogno
- Si può avere **frammentazione interna** però solo a livello dell'ultima pagina che costituisce il file
- L'unico problema è il **dimensionamento del blocco indice**:
  - se è troppo piccolo il file potrebbe richiedere più blocchi dati quanti è possibile riferire
  - se è troppo grande si ha frammentazione interna al blocco indice stesso (spreco di memoria)
  - **soluzione ideale**: poter ridimensionare il blocco indice a seconda delle circostanze ...

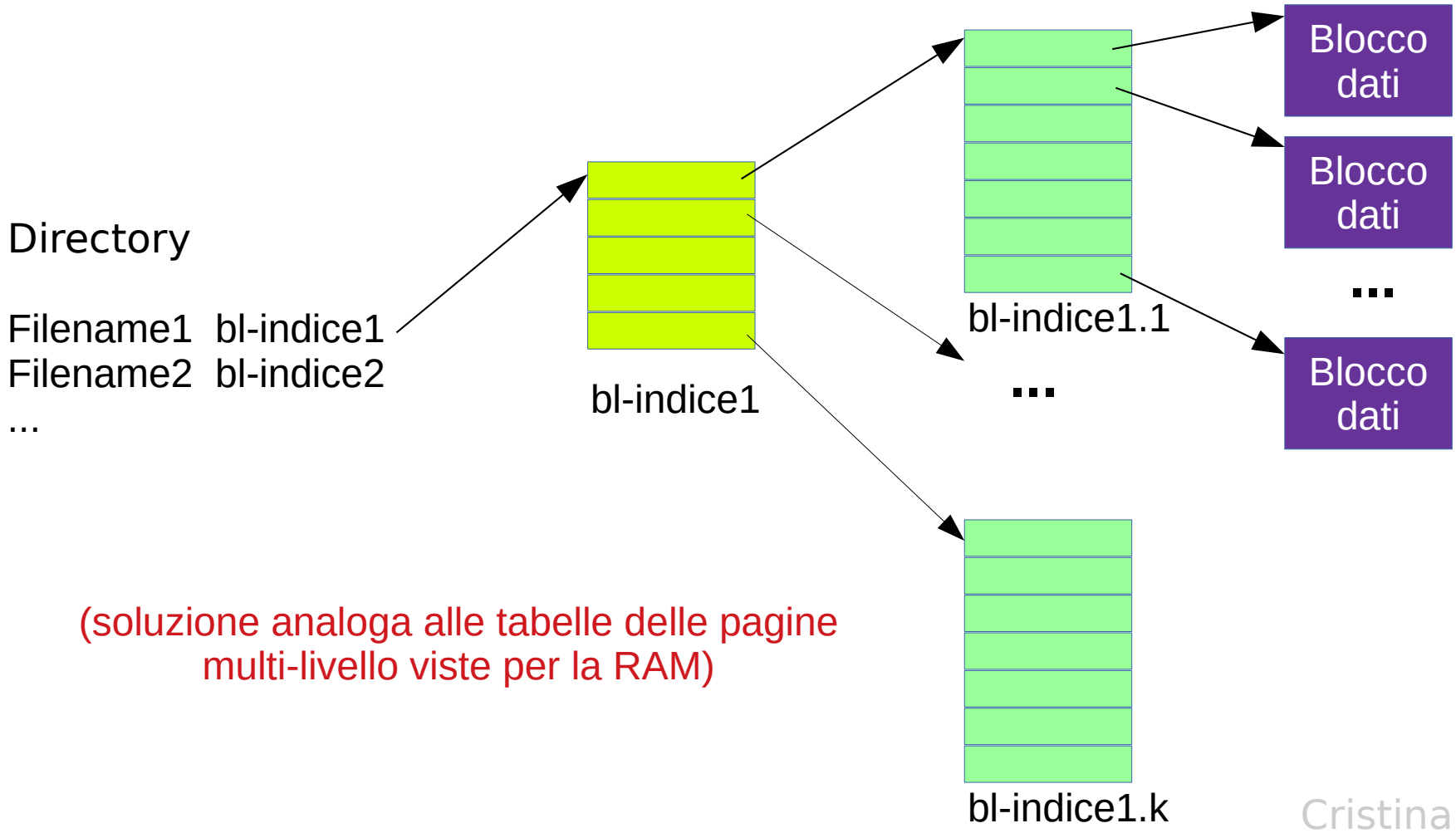
Implementazione a schema concatenato

Utilizzare blocchi indice piuttosto piccoli e interpretare l'ultimo indirizzo contenuto nel blocco indice come riferimento a un'estensione del medesimo, da usare solo se serve



Implementazione con indice a più livelli

Struttura gerarchica. Il primo livello punta a una serie di blocchi indice, ciascuno dei quali consente l'accesso a blocchi dati. I blocchi indice/dati vengono aggiunti solo se serve

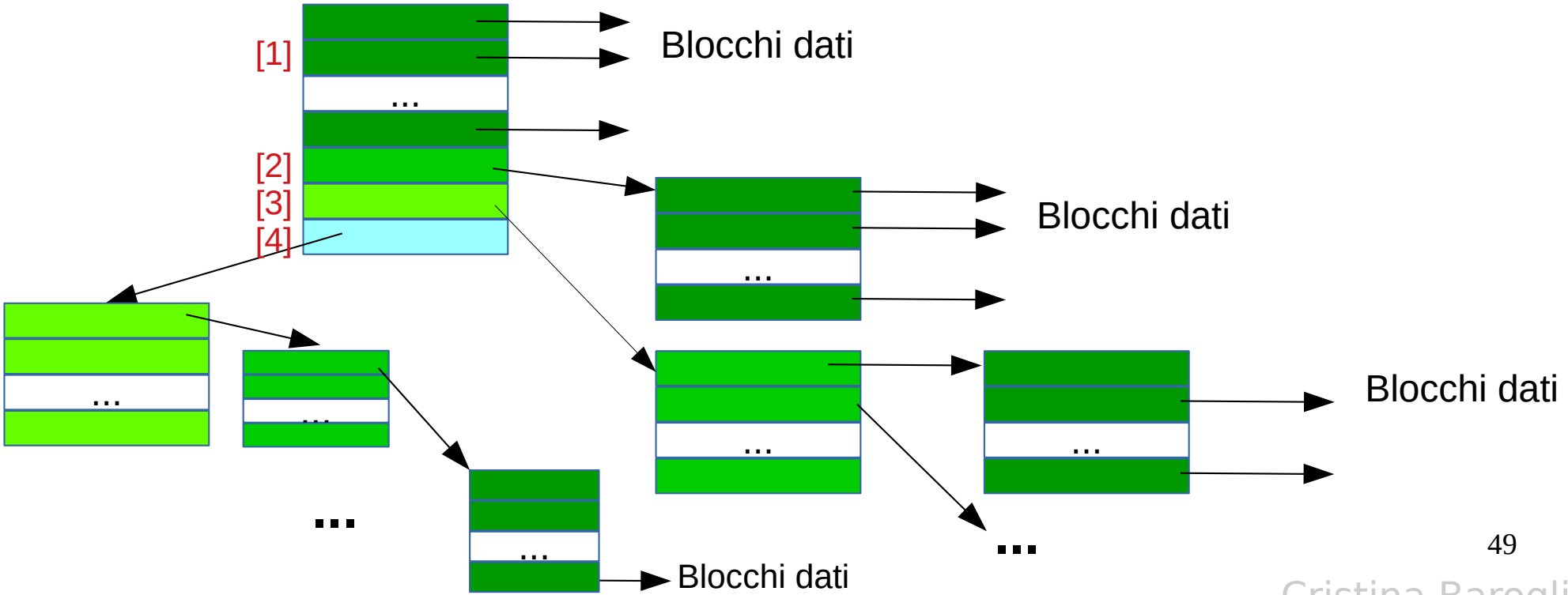


(soluzione analoga alle tabelle delle pagine multi-livello viste per la RAM)

Soluzione ibrida

Viene mantenuta una tabella di accesso ai dati, una parte dei cui elementi consente l'accesso diretto a blocchi mentre un'altra parte consente l'accesso attraverso un indice a più livelli

Soluzione adottata in Unix (per esempio), dove la tabella è inclusa nell'inode



Soluzione ibrida

[1] ogni entry è un riferimento a un blocco dati

[2] ogni entry è un riferimento a una tabella di riferimenti a blocchi dati

[3] ogni entry è un riferimento a una tabella di riferimenti a una tabella di riferimenti a blocchi dati

[4] ogni entry è un riferimento a una tabella di riferimenti a una tabella di riferimenti a una tabella di riferimenti a blocchi dati

si hanno da 15 a 19 blocchi a indirzzamento diretto [2], [3] e [4] sono allocati solo se serve

Soluzione ibrida

Vantaggio

- con questa tecnica si possono costruire file che raggiungono dimensioni molto grandi (terabyte)

Svantaggio

- laddove si usino i livelli di indicizzazione indiretta questa tecnica soffre un po' dei limiti della tecnica di concatenazione

Commenti generali

- **Allocazione concatenata**: più adeguata a accessi sequenziali
- **Allocazione a indice**: più adeguata ad accessi diretti
- L'allocazione indicizzata richiede di mantenere in RAM una parte dei blocchi indice, possono occorre due o più accessi al disco se la RAM non è sufficiente:
  - uno (o più) per accedere al blocco indice giusto
  - uno per raggiungere il dato di interesse
- Alcuni sistemi combinano allocazione contigua e indicizzata: finché il file rimane di piccole dimensioni si usa l'allocazione contigua, oltre un certo limite si comincia ad usare un indice

Gestione dello spazio libero

- Occorre una struttura che consente di tener traccia dei blocchi liberi
- **Vettore di bit**: viene mantenuto un array di bit, ognuno dei quali corrisponde a un blocco. Se il blocco è libero il bit è impostato a 1, es:

  0 0 0 1 1 0 1 1 1 1 0 1 0 1 1 1 0 0 0 1 1 0

  il quarto e quinto blocco sono liberi, idem dal settimo al decimo e così via

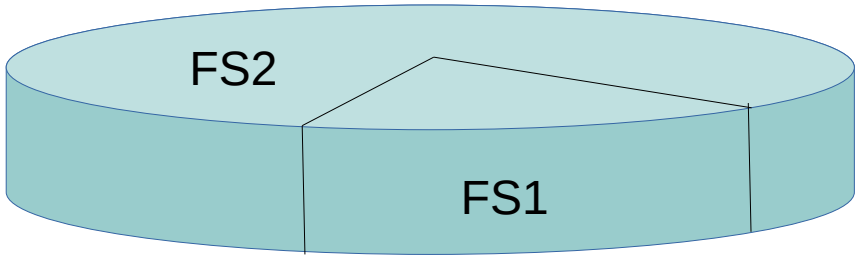
- NB: con questa soluzione è anche facile identificare sequenze di blocchi contigui liberi. Se so che mi servono tre blocchi liberi e li trovo contigui, meglio, perché i tempi di accesso saranno inferiori

Gestione dello spazio libero

- **Lista concatenata**: i blocchi liberi sono concatenati in una lista. Poiché i blocchi vengono allocati ai file uno per volta, basta pescare dalla testa della lista il primo blocco libero ed aggiornare il puntatore
- **Raggruppamento**: concatenazione di blocchi indice. Si usa un blocco libero (di dimensione N) per mantenere N-1 puntatori ad altrettanti blocchi liberi. Si usa l'ultimo puntatore per fare riferimento a un eventuale ulteriore blocco dello stesso tipo
- **Conteggio**: variante del precedente, in presenza di sequenze di blocchi liberi contigui si mantiene un riferimento al primo di tali blocchi e il numero di blocchi liberi ad esso consecutivi. Es. <K, 4> dove K è il riferimento a un blocco libero e 4 il numero di blocchi liberi consecutivi a partire da esso

Quantità di memoria massima gestibile

- La quantità massima di memoria gestibile da un file system dipende dalle strutture del file system
- **Esempio**: una FAT a 8bit consente di indicizzare al più 2 8 blocchi (256 blocchi), se ogni blocco è grande 1KB potrò al più gestire una memoria pari a 256KB
- Se devo gestire un disco di dimensioni superiori alla quantità di memoria gestibile da un FS dovrò partizionare il disco in FS diversi o una parte della memoria sarà inaccessibile!!

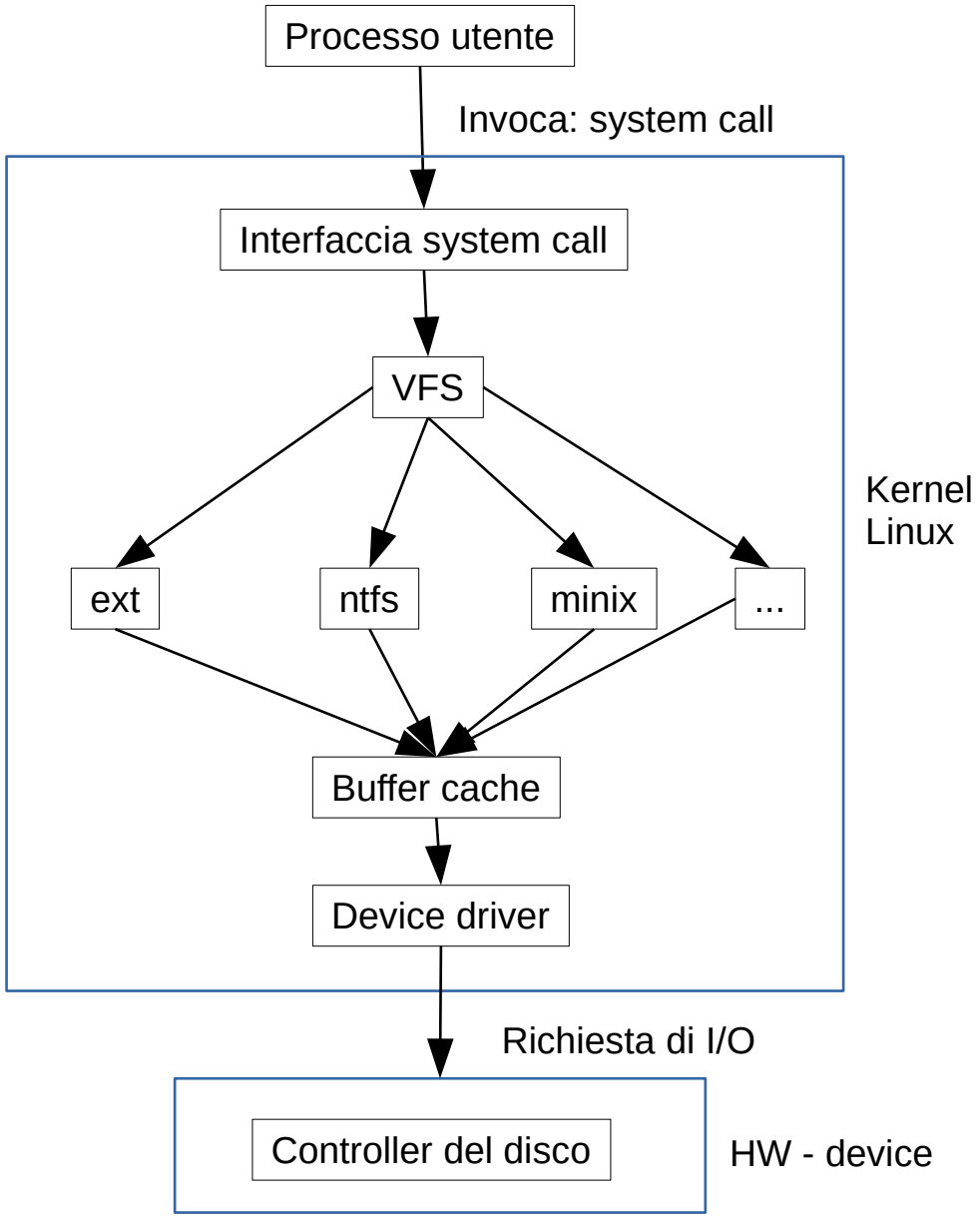


File System Virtuale

- Un disco può essere partizionato e ogni partizione può contenere FS di tipodifferente, eppure è possibile montare i diversi FS in un'unica struttura a cui l'utente ha accesso attraverso lo stesso insieme di comandi e system call, senza accorgersi delle diverse implementazioni
- Analogamente è possibile accedere a FS di rete senza rendersi conto che i file usati risiedono su una macchina diversa da quella in uso

- Ciò è possibile perché l'interazione fra i processi e il FS è mediata da un'**interfaccia astratta** che permette di prescindere dallo specifico tipo di FS usato
- Questa interfaccia è il **Virtual File System**

File system virtuale



I processi utente invocano delle system call offerte dal VFS

Il VFS sta a un livello di astrazione più alto dei FS veri e propri ed esegue quella parte del lavoro che è indipendente dalla realizzazione fisica

Il VFS è un livello di indirezione che gestisce le sys. call che lavorano su file, preoccupandosi di richiamare le necessarie routine dello specifico FS fisico per attuare l'I/O richiesto

I diversi FS fisici devono specificare l'implementazione di tutte le sys. call offerte dal VFS

Oggetti di un VFS

- Il VFS gestisce **vnode**:
  un vnode è un'**astrazione degli inode**, rappresenta un file, una directory, un link, un socket, un block o un character device, ecc.
- **NB**: mentre un inode ha un identificatore numerico unico per uno specifico FS, un vnode ha un id numerico unico per tutto l'insieme dei FS montati
- I vnode possono essere di diversi tipi a seconda di ciò che rappresentano, per esempio:
  - **superblock**: rappresenta un intero FS
  - **dentry**: rappresenta un generico elemento di una directory
  - **inode**: rappresenta un generico file
  - ...

Operazioni consentite da un VFS

- Un VFS definisce e mette a disposizione del livello utente un insieme di operazioni (**system call**) di base, fra cui troviamo per esempio:
  - open di un file
  - close di un file
  - read/write di un file
- uno specifico FS fisico, es. ext3 o winFS, per essere gestibile attraverso il VFS deve implementare tali funzioni, esattamente come accade per le interfacce in un linguaggio object-oriented.
- L'implementazione è talvolta indicata con il termine: **driver del file system**

Buffer-cache?

- Occorre fare una distinzione fra due tipi di memorie d'appoggio, definiti in base all'uso che se ne fa:
  - **buffer**: è una memoria in cui i dati vengono inseriti in attesa di essere consumati. I dati in un buffer vengono usati una volta sola.
  - **cache**: è una memoria in cui i dati vengono conservati per un po' di tempo in previsione di ulteriori utilizzi

- I dischi hanno associata una **memoria di appoggio in RAM** detta **buffer cache** ...

Buffer Cache

- **Passi di una scrittura su file**:
  - **Processo**: esegue una **write**
  - Il blocco da modificare, appartenente al file in questione, viene aggiornato **in RAM**
  - La modifica viene **propagata** al blocco del file conservato su disco
- Normalmente un'operazione di scrittura su di un file si conclude solo quando la modifica su disco è stata completata
- **Write-ahead**: meccanismo tramite il quale un'operazione di scrittura viene considerata conclusa al termine dell'aggiornamento della copia del blocco *contenuta in RAM*

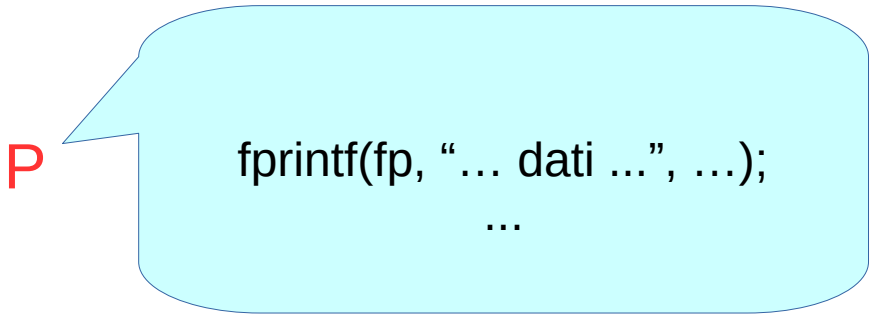
Read ahead – read behind

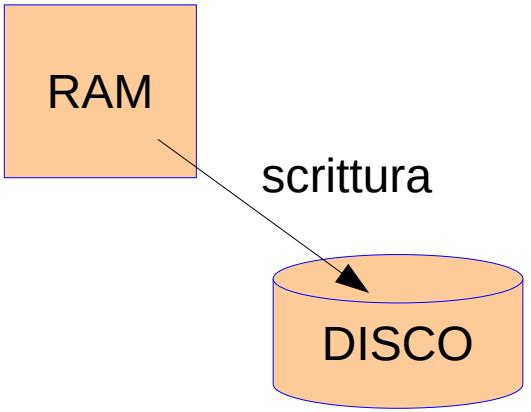
Quando il controller riceve una richiesta di lettura, individua un settore del disco e poi lo legge tutto (non solo il dato richiesto). La parte extra è inserita nella RAM e lasciata a disposizione per usi futuri.

Questo approccio applica una sorta di criterio di località negli accessi al disco è un'attività nota come **read ahead** o **read behind**

Buffer Cache

- **Perché il write ahead incrementa l'efficienza?**
- Consideriamo un processo che esegue la scrittura senza write ahead:

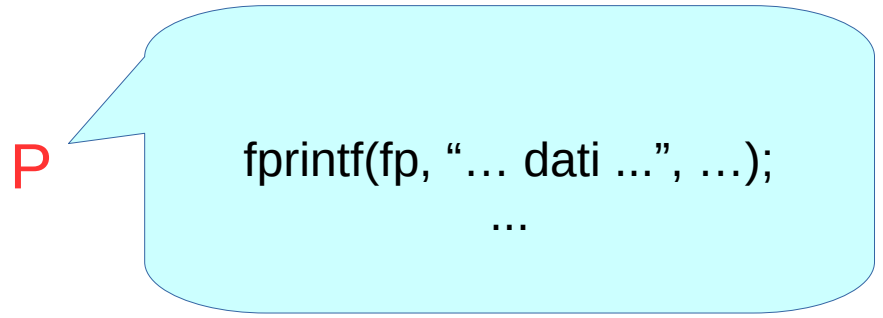


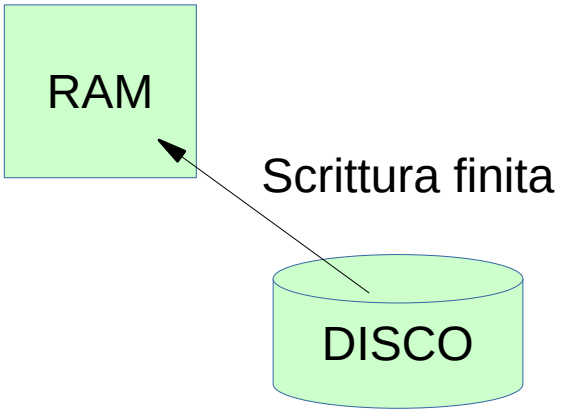


- **Per eseguire questo codice il processo è nello stato RUNNING**
- **La scrittura è un'operazione di output, come tutte le operazioni di I/O fa cambiare lo stato del processo in WAITING**

Buffer Cache

- **Perché il write ahead incrementa l'efficienza?**
- Consideriamo un processo che esegue la scrittura senza write ahead:

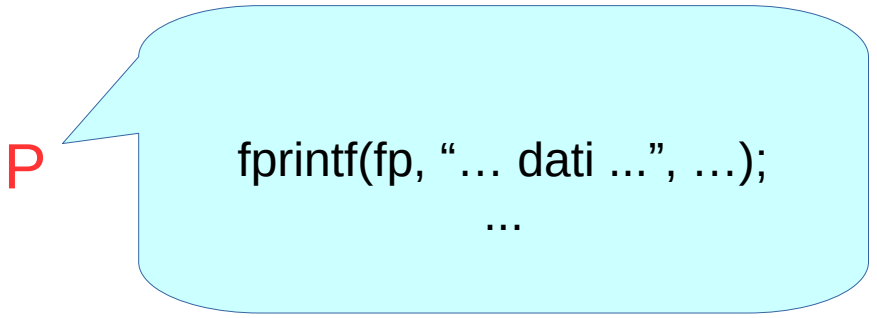




- **P rimane WAITING finché non termina la scrittura su disco**
- **Dopo P diventa READY**
- **Infine dopo un tempo che dipende dallo scheduling della CPU, P torna RUNNING e prosegue la sua esecuzione**

Buffer Cache

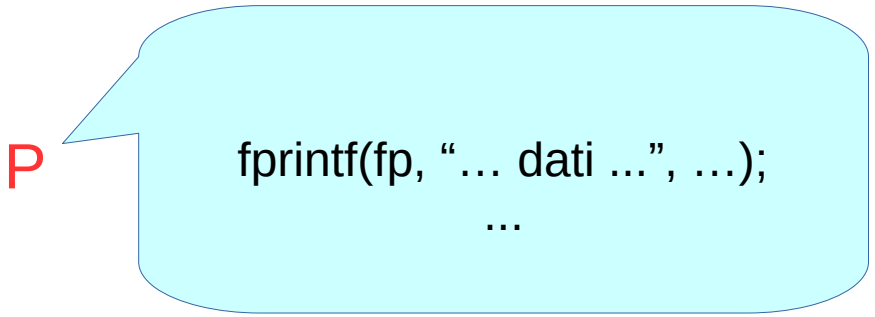
- **Perché il write ahead incrementa l'efficienza?**
- Consideriamo lo stesso processo che esegue la scrittura con il write ahead:

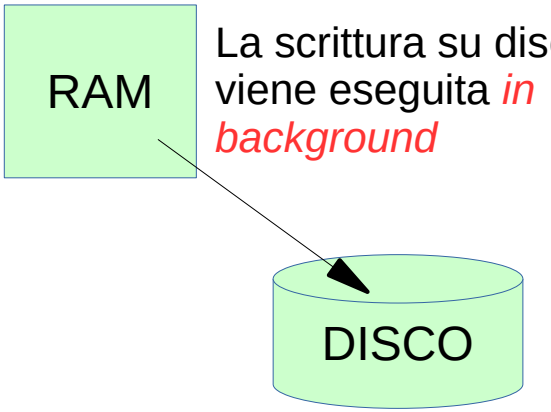


  - **Per eseguire questo codice il processo è nello stato RUNNING**
  - **Poiché si usa il write-ahead, La scrittura è considerata terminata quando è completata in RAM**
  - **Il processo continua la sua esecuzione senza cambiare stato**

Buffer Cache

- **Perché il write ahead incrementa l'efficienza?**
- Consideriamo lo stesso processo che esegue la scrittura con il write ahead:



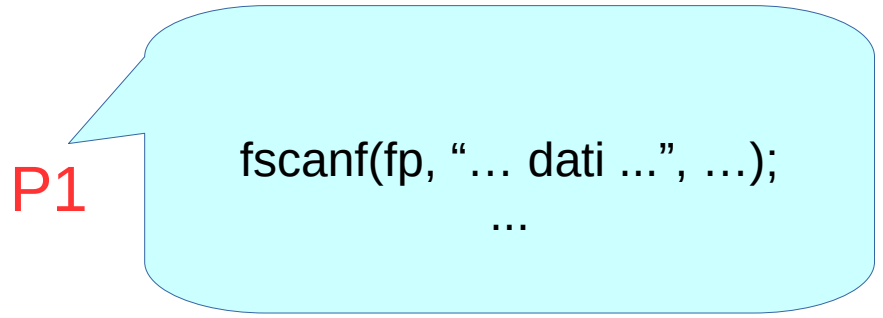


- **Per eseguire questo codice il processo è nello stato RUNNING**
- **Poiché si usa il write-ahead, La scrittura è considerata terminata quando è completata in RAM**
- **Il processo continua la sua esecuzione senza cambiare stato**

È necessario che l'interazione di scrittura con il device non richieda l'intervento della CPU (es. DMA)

Buffer Cache

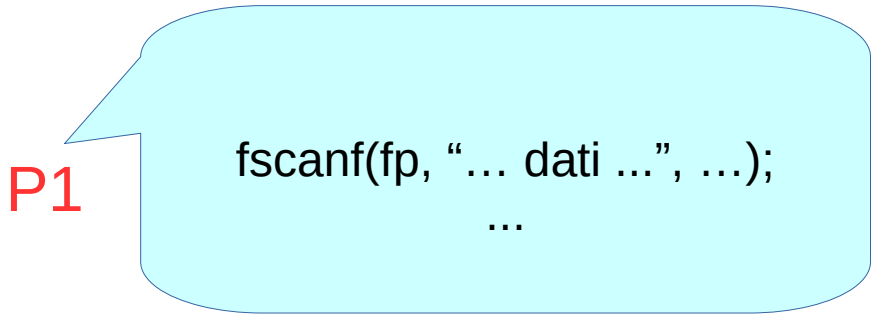
- **Perché il buffer cache incrementa l'efficienza?**
- Consideriamo un processo che voglia leggere il file prima scritto da P:



  - **Per eseguire questo codice il processo è nello stato RUNNING**
  - **La lettura è un'operazione di input, dovrebbe far cambiare lo stato del processo a WAITING**

Buffer Cache

- **Perché il buffer cache incrementa l'efficienza?**
- Consideriamo un processo che voglia leggere il file prima scritto da P:



  - **Poiché si usa il buffer cache, i blocchi dati modificati sono in RAM**
  - **Il processo vi accede e continua la sua esecuzione senza cambiare stato**

Esempio

- Un utente scrive il programma **codice.c** usando **editor di testo** e **compilatore**. Supponiamo che l'utente modifichi il file, lo salvi e chiuda l'editor e solo dopo compili il file modificato:
  - L'**editor** è un esempio di processo che scrive un file
  - senza write ahead, ad ogni *save* l'editor rimarrebbe WAITING fino al termine della copiatura su disco
  - Con il write ahead a ogni save l'editor consente all'utente di continuare a lavorare non appena la richiesta di scrittura è stata notificata al controller del disco
  - La scrittura su disco vera e propria avverrà in background

Esempio

- Un utente scrive il programma **codice.c** usando **editor di testo** e **compilatore**. Supponiamo che l'utente modifichi il file, lo salvi e chiuda l'editor e solo dopo compili il file modificato:
- Il **compilatore** è un esempio di programma che legge un file per elaborarlo. Il file risiede su disco ma la presenza del buffer cache rende più efficiente l'accesso ai blocchi dati:
  - se **codice.c** è appena stato salvato, anche se è stato chiuso dal processo editor, i suoi blocchi dati sono (probabilmente) ancora nel buffer cache
  - non serve aspettare il caricamento del file da disco per effettuare la compilazione

Ripristino del file system

- Quando si avvia un elaboratore, il SO compie una verifica della **consistenza del file system.** **FSCK** (**file system check**): è una routine che controlla se all'ultimo spegnimento il FS è stato smontato (operazione di unmount) correttamente.
- L'**unmount** forza l'esecuzione delle scritture pendenti

- Il modo in cui viene effettuato il controllo dipende dal FS. Es. in ext2 si verifica un campo del superblocco, che può assumere i valori **clean** e **unclean**

1) quando si fa il mount del FS il campo è settato ad unclean
2) quando si fa l'unmount viene settato a clean
3) se c'è un crash, il campo rimane unclean

Ripristino del file system

- Se fsck scopre che il FS non è stato smontato correttamente, avvia un controllo minuzioso (che non studiamo) del disco intero, individuando le inconsistenze e risolvendole per quanto possibile
- **La durata di questo controllo dipende dalla dimensione del disco**
- **PROBLEMI**:
  - Se il problema si verifica su un server che mantiene i dati di qualche servizio critico (es. banca, compagnia di volo, ...) il tempo trascorso nella verifica è un costo perché durante tutto il tempo il FS è off-line e non può essere usato
  - altro problema: transazioni. Se il crash ha interrotto una transazione, occorrerà avviare delle operazioni di ripristino di uno stato consistente

- **Soluzione**: mantenere qualche informazione in più che consenta di andare ad effettuare verifiche mirate; perché controllare porzioni di file modificate un'ora fa? Soltanto le modifiche effettuate poco prima del crash possono non essere state applicate ...

Journal

- Il **journal** è un **logfile** mantenuto su disco, in cui si tiene nota di tutte le operazioni di scrittura su file
- viene gestito in modo simile ai logfile per le transazioni quando un processo effettua una scrittura, prima il SO registra l'operazione che si va ad effettuare sul logfile, poi esegue la scrittura vera e propria del dato su file

Journaling del file system

- Le annotazioni presenti nel **journal** consentono di focalizzare il recovery sui soli file su cui sono state effettuate scritture **negli ultimi istanti prima del crash**
- Prima viene scritta l'annotazione nel journal, poi viene modificato il file oggetto della scrittura
- Consente di distinguere fra i seguenti casi, a seconda del momento in cui è avvenuto il disastro da recuperare:
  - **Crash successivo alla scrittura su disco del file**: l'annotazione nel journal corrisponde a quanto presente nel file, nessuna azione viene intrapresa
  - **Crash occorso fra la scrittura del journal e la scrittura su disco del file**: l'annotazione nel journal non corrisponde al contenuto del file, il contenuto del file viene reso consistente rispetto all'annotazione
  - **Crash occorso prima dell'annotazione nel journal**: la scrittura non ha mai avuto luogo

Journaling

Vantaggio

- La durata del ripristino non dipende dalle dimensioni del FS, bastano pochi secondi per riportare la consistenza dei dati

Svantaggio

- La gestione del journal appesantisce la gestione delle operazioni sui file. Nonostante ciò oggi come oggi i journal sono considerati essenziali