

File-System Implementation

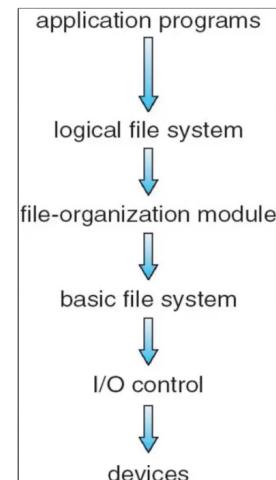
Overview

File System Structure

- Disks provide most of the secondary storage on which file systems are maintained.
 - A disk can be rewritten in place. It is possible to read a block from the disk, modify the block, and write it back into the same block.
 - A disk can access directly any block of information it contains.

Layered File System

- File system is composed of many different levels:
 - I/O control
 - Basic file system
 - File-organization module
 - Logical file system
 - Application programs



Un filesystem è composto da più "livelli": i **Dispositivi** rientrano nel livello delle tecnologie usate per memorizzare i dati, tutti gli altri livelli invece sono "inclusi" all'interno del Sistema Operativo.

- **I/O Control**

Livello in cui troviamo i **device drivers**, che si occupano di comunicare coi dispositivi.

- **Basic file system**

Livello che consente di impartire i **comandi** ai device drivers (leggi, scrivi etc..).

Questi comandi arrivano al device driver grazie a questo livello di basic file system.

- **File-organization module**

Livello che contiene le informazioni riguardo l'implementazione logica dei dati, quindi ad esempio quanti blocchi logici occupa un file.

- **Logical file system**

Viene usato per gestire i metadati dei dati, in questo livello troviamo un modulo chiamato **File Control Block**, che contiene, per ciascun file, tutti i metadati.

Ad esempio quando creiamo un nuovo file, questo livello crea un nuovo File Control Block in cui inserisce i metadati del file da creare, e poi scendendo ai livelli inferiori si vanno a fare le operazioni necessarie per creare il file.

- Managing **metadata** information including all of the file-system structure except the contents of the files.
- Managing the directory structure to provide the file-organization module with the information needed.
- Maintaining file structure via **file-control blocks** containing information about the file, such as ownership, permissions and location of the file contents.

Logical file system

Layered File System

- Benefits of layer structure:
 - Duplication of code is minimized. The I/O control and sometimes the basic file-system code can be used by multiple file systems.
 - Each file system can have its own logical file-system and file-organization modules.
- Disadvantages:
 - Layering can introduce more operating-system overhead, which may result in decreased performance

Il vantaggio di avere un file system a livelli è che alcuni di questi possono essere *condivisi* da file system diversi. Ad esempio potrei avere FAT32 e NTFS su uno stesso sistema operativo, ed entrambi i file system possono sfruttare il medesimo livello di basic file system.

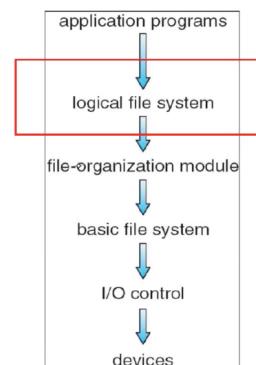
Per lo stesso motivo però, ciò risulta in un overhead maggiore e dunque performance più basse di un sistema unico "ad hoc" — pensiamo ad esempio se un file system volesse fare un'operazione di I/O ma si trova a dover aspettare perché l'

I/O control è in uso da un altro file system al momento.

In-Memory File System Structures

- To create a new file, a process calls the logical file system.
 - The logical file system knows the format of the directory structures.
 - To create a new file, it allocates a new FCB.
 - The system then reads the appropriate directory into memory, updates it with the new file name and FCB and writes it back to the file system.

file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks or pointers to file data blocks



Allocation Methods

Allocation Methods

- An allocation method refers to how disk blocks are allocated for files:
 - Contiguous allocation
 - Linked allocation
 - Indexed allocation

Come facciamo a decidere quali blocchi allocare per il file da creare?

Ci sono 3 modi.

Contiguous Allocation

Allocation Methods - Contiguous

- An allocation method refers to how disk blocks are allocated for files:
- **Contiguous allocation** – each file occupies set of contiguous blocks
 - Best performance in most cases
 - Simple – only starting location (block #) and length (number of blocks) are required
 - Accessing file:
 - For sequential access, the file system remembers the address of the last block references and read the next block.
 - For direct access to block i of a file that starts at block b , we can immediately access block $b+i$.

Contiguous Allocation

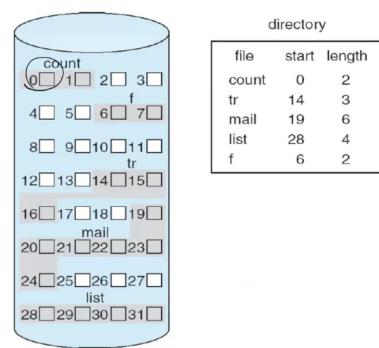
- Mapping from logical to physical

LA/512

Q

R

Block to be accessed = Q +
starting address
Displacement into block = R



Extent-Based Systems

- **Problems of contiguous allocation:**
- Suffering from external fragmentation.
 - Files are allocated and deleted; the free storage space is broken into little pieces.
 - Largest amount of contiguous memory is not sufficient for a request.
 - **Solution:** copy an entire file system onto another device and copying the file back onto the original device by allocating contiguous space from this one large hole: **compacting** all free space into one large hole.



Svantaggi: possiamo accedere alle informazioni in modo sequenziale o diretto. Se accediamo in modo sequenziale o diretto, l'allocazione contigua può andare bene.

Però potremmo incappare in
frammentazione esterna!

Un modo per risolvere sarebbe la **deframmentazione del disco**, che va a compattare gli spazi che si sono creati, ma è un'operazione costosa.

Extent-Based Systems

- **Problems of contiguous allocation:**
- Determining how much space is needed for a file.
 - When a file is created, the total amount of space it will need must be found and allocated which is difficult to estimate.
 - If we allocate too little space to a file, we may find the file cannot be extended.
Solutions:
 1. The user program can be terminated with an error message and run the program again.
 2. Finding a larger hole, copying the contents of the file to the new space and release the previous space.
- Time Consuming Solutions

Con l'allocazione contigua abbiamo problemi anche nel caso in cui volessimo estendere lo spazio dedicato ad un file.

Extent-Based Systems



- **Problems of contiguous allocation:**

- Determining how much space is needed for a file.
 - Modified contiguous-allocation scheme:
 - A contiguous chunk of space is allocated initially.
 - If the amount proves not to be large enough, another chunk of contiguous space, known as an **extent** is added.
 - The location of a file's blocks is then recorded as a location and a block count, plus a link to the first block of the next extent.

Per questo motivo esistono un altro tipo di allocazione di blocchi: **Extent-Based Systems**.

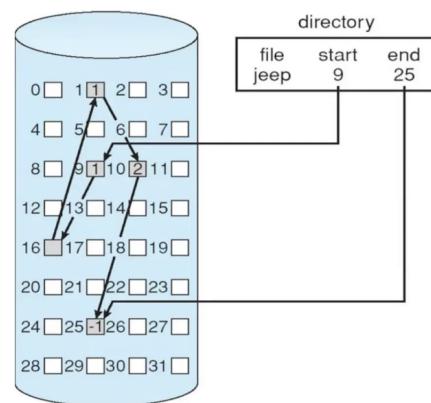
Questi sistemi ci consentono di dedicare inizialmente un certo numero di blocchi, e se necessario poi estenderli tramite altri blocchi, non necessariamente contigui. Ciò viene fatto memorizzando, nell'ultimo blocco dei primi blocchi allocati per il file, la posizione dei nuovi blocchi allocati.

Linked Allocation

Allocation Methods - Linked

- **Linked allocation** – each file a linked list of storage blocks

- Each block contains pointer to next block
- Reliability can be a problem
- Locating a block can take many I/Os and disk seeks
- Disadvantages: it can be used only for sequential access files. To find the i^{th} block of file, we must start from the beginning of the file.



In ogni blocco è memorizzato l'indirizzo del blocco successivo.

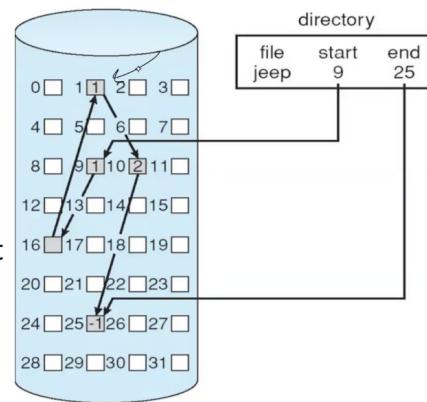
Svantaggi: non possiamo avere accesso diretto ai blocchi — se volessimo ad esempio accedere al quarto blocco, non sapremmo a priori dove si trova.

Dovremo per forza percorrere tutto il percorso fino al quarto blocco.

Vantaggi: non abbiamo frammentazione esterna. Non abbiamo problemi se vogliamo estendere il file (basta trovare un blocco libero e aggiornare i metadati riguardo l'**end** del file).

Allocation Methods - Linked

- **Linked allocation** – each file a linked list of storage blocks
 - File ends at nil pointer
 - No external fragmentation
 - **Each block contains pointer to next block**
 - No compaction, external fragmentation
 - Free space management system called when new block needed
 - Improve efficiency by clustering blocks into groups but increases internal fragmentation
 - Reliability can be a problem
 - Locating a block can take many I/Os and disk seeks
 - Disadvantages: it can be used only for sequential access files. To find the i^{th} block of file, we must start from the beginning of the file.



Altro svantaggio: in ogni blocco devo riservare dello spazio per memorizzare le informazioni riguardo il blocco successivo — è un po' uno spreco di spazio.

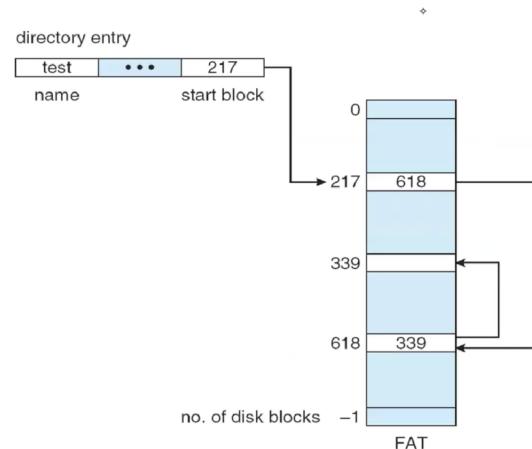
Una soluzione è il

clustering: sappiamo che di solito quando ho un file ho bisogno di più blocchi per esso, quindi ne "raggruppo" ad esempio 4, e questo gruppo lo chiamo *cluster*. In questo modo invece di avere 4 puntatori (uno per ciascun blocco), ne ho uno solo.

Allocation Methods - Linked

- An important variation on linked allocation is the use of a **File Allocation Table (FAT)** variation

- A section of storage at the beginning of each volume is set aside to contain the table.
- The table has one entry for each block and is indexed by block number.
- The directory entry contains the block number of the first block of the files.
- The table entry indexed b that block number contains the block number of the next block in the file.
- This chain continues until it reached the last block which has a special end-of-file value as the table entry.



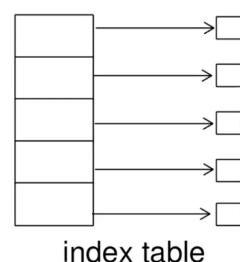
Un'altra soluzione a questo problema è l'uso di una **File Allocation Table**.

Tale tabella memorizza la posizione dei blocchi allocati per il file.

Indexed Allocation

Allocation Methods - Indexed

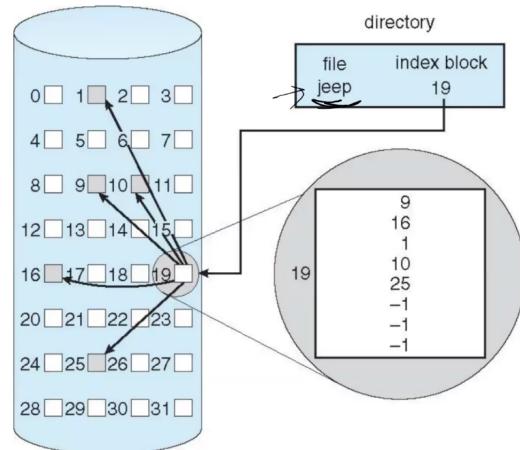
- Linked allocation solves the external-fragmentation and size-declaration problems of contiguous allocation.
- In the absence of a FAT, linked allocation cannot support efficient direct access.
- Indexed allocation** solve this problem by bringing all the pointers together into one location: the **index block**
 - Each file has its own index block(s) of pointers to its data blocks



È il metodo su cui si basano più comunemente i sistemi Unix.

Example of Indexed Allocation

- Each file has its own index block which is an array of storage-block addresses.
- The i^{th} entry in the index block points to the i^{th} block of the file.
- The directory contains the address of the index block.
- To find and read the i^{th} block, we use the pointer in the i^{th} index-block entry.



Vantaggi: abbiamo accesso diretto, perchè basta leggere l'index block per trovare la posizione di tutti i blocchi di dati.

Svantaggi: se il file è piccolo, sprechiamo spazio nell'index block, che rimane "vuoto".

Indexed Allocation

- How large the index block should be?
 - Every file must have an index block.
 - The index block should be as small as possible.
 - If too small, it will not be able to hold enough pointers for a large file.
- To solve this issue:
 - Combined Scheme

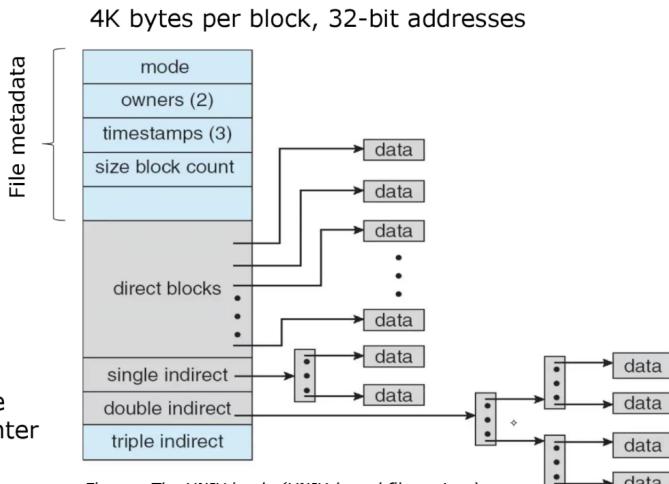
La soluzione più efficiente è usare una combinazione di queste tecniche: **Unix UFS**.

Unix UFS

Combined Scheme: UNIX UFS

- Direct Blocks
- Indirect blocks
 - Single indirect blocks
 - Double indirect blocks
 - Triple indirect blocks

More index blocks than can be addressed with 32-bit file pointer



Se ho un file piccolo non usiamo l'Indexed Allocation, ma scriviamo direttamente l'indirizzo dei suoi blocchi nella sezione **direct blocks**.

Se il file è più grande, usiamo la modalità successiva di accesso:

single indirect — cioè scriviamo in questa sezione l'indirizzo di un singolo blocco, associato al file, che memorizza tutti i puntatori ai blocchi di dati (essenzialmente è un *index block*).

Ovviamente tale blocco ha una dimensione

finita: se ad esempio avessimo una dimensione di blocco di 4KB, e avessimo indirizzamento a 32bit, vorrebbe dire che potremmo scrivere 4B di informazioni in ogni "riga" del blocco. Dunque $4KB/4B \Rightarrow 1024$ entries, cioè l'index block può puntare a 1024 blocchi dati.

Dunque se dovessimo memorizzare un file che ha bisogno di più di 1024 blocchi dati, potremmo usare la modalità successiva:

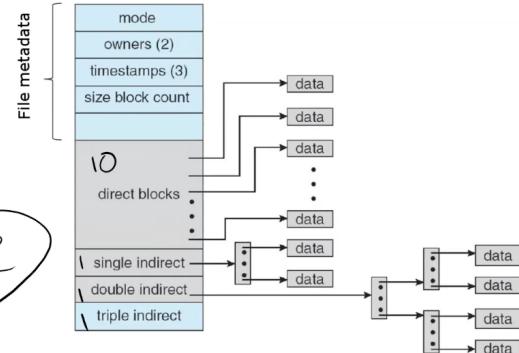
double indirect — in cui l'index block punta ora ad altri *index blocks* (quindi potenzialmente 1024 index blocks diversi, con l'esempio precedente), ciascuno dei quali punta a dei blocchi dati (potenzialmente 1024). Dunque in questo caso potremmo memorizzare $1024 * 1024 = 1.048.576$ blocchi di dati.

C'è poi, ancora, il

triple indirect che introduce un terzo livello di index blocks.

Combined Scheme: UNIX UFS

- UNIX-based file system: 15 pointers of the index block in the file's inode.
 - Direct Blocks : 12 pointers points to direct blocks. They contains addresses of blocks that contain data of the file.
- Indirect blocks: three pointers points to indirect blocks.
 - **Single indirect blocks**: an index block containing not data but the addresses of blocks that contain data.
 - **Double indirect blocks**: contains the address of a block that contains the addresses of blocks that contain pointers to the actual data blocks.
 - **Triple indirect blocks**: contains the address of a triple indirect block.



IMPORTANTE PER ESAME

Free-space Management

- ### Free-space management
- Since storage space is limited, we need to reuse the space from deleted files for new files.
 - File system maintains **free-space list** to track available blocks/clusters.
 - To create a file:
 - Search the free space list for the required amount of space and allocate that space to the new file.
 - The space is then removed from the free-space list.
 - When the file is deleted, its space is added to the free-space list.

Abbiamo parlato finora di tecniche per allocare blocchi dati ai files.

Però dobbiamo in qualche modo sapere

quali blocchi disponibili ci sono sul disco.

Solitamente il SO tiene una

free-space list che tiene traccia dei blocchi liberi.

Ci sono 3 modi per capire quali sono i blocchi liberi.

Free-space management

- The free-space list is implemented as
 - bitmap or bit vector
 - Linked list
 - Grouping
 - Counting

Bitmap o bit vector

Free-space management

- The free-space list is implemented as **bitmap** or **bit vector**.
- Each block is represented by 1 bit.
 - If the block is free, the bit is 1. if the block is allocated, the bit is 0.
- For example, consider a disk where blocks 2,3,4,5,8,9,10,11,12,13,17,18,25,26, and 27 are free and the rest of the blocks are allocated. The free space bitmap would be:
 - 001111001111110001100000011100000...
- Advantages: Simplicity and efficiency in finding the first free block

A ogni blocco associamo un bit. Se tale bit è 1 → il blocco è libero.

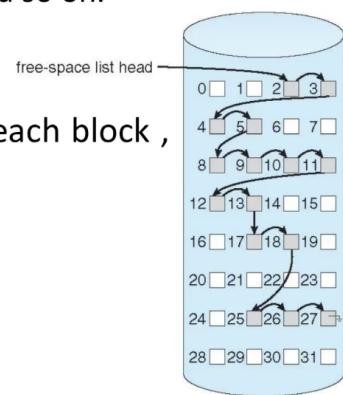
Free-space management

- Benefits: its relative simplicity and its efficiency in finding the first free block or n consecutive free blocks on disc.
+
• Drawbacks: inefficient unless the entire vector is kept in main memory. Keeping it in memory is possible for smaller devices but not necessarily for larger ones.
- A 1.3 GB disk with 512-byte blocks would need a bitmap over 332KB to track its free blocks, although clustering blocks in groups of four reduces this number to around 83KB per disk.
- A 1TB disk with 4KB blocks would require 32MB to store its bitmap. ($2^{40}/2^{12} = 2^{28}$ bits = 2^{25} bytes = 2^5 MB)

Linked Free Space List on Disk

Linked Free Space List on Disk

- Another approach to free-space management is to link together all the free blocks, keeping a pointer to the first free block in a special location in file system and caching it in memory.
- The first block contains a pointer to the next free block and so on.
- Not efficient since to traverse the list, we must read each block , which requires substantial I/O times on HDDs.



Tutti gli spazi liberi che abbiamo sono *linkati*, dunque memorizziamo solo l'indirizzo del primo blocco disponibile, e poi ciascun blocco punta al prossimo blocco libero.

Grouping e Counting

Free Space Management

- Grouping
 - Modify linked list to store address of next $n-1$ free blocks in first free block, plus a pointer to next block that contains free-block-pointers (like this one)
- Counting
 - Because space is frequently contiguously used and freed, with contiguous-allocation allocation, extents, or clustering
 - Keep address of first free block and count of following free blocks
 - Free space list then has entries containing addresses and counts

Esercizi

Esercizio 1

Exercise 1:

- Siano dati due file system F1 e F2, su due volumi, basati rispettivamente su FAT e su Inode. I **puntatori hanno dimensione di 32 bit**, i blocchi hanno dimensione 4KB, entrambi i volumi hanno una parte riservata ai metadata (direttori, FAT e FCB –file control block) oppure Inode, blocchi indice, ecc) e una parte ai blocchi di dato. Si supponga che lo spazio riservato ai blocchi di dato sia lo stesso per F1 e F2. Si sa che la **FAT occupa 150 MB**.

A) Quanti blocchi di dato possono contenere complessivamente, al massimo, i file presenti nei due file system?

- **FAT (File Allocation Table)** uses a file allocation table to keep track of the location of each file on the storage device.
- **An inode (short for "index node")** is a data structure used by most file systems in operating systems to store information about a file or directory.

Dunque:

- Puntatore = 32bit = 4B
- Block size = 4KB
- FAT size = 150MB

Siccome ogni entry nella FAT rappresenta un blocco, tramite puntatore a 32bit (4B), si ha che: $150MB/4B = 37,5M$ blocchi (cioè 39.321.600 blocchi, perchè $37,5 \times 1024 \times 1024$).

Sappiamo che F1 ed F2 hanno lo stesso numero di blocchi di dati, dunque anche anche F2 contiente 37,5M blocchi.

Exercise 1:

Siano dati due file system F1 e F2, su due volumi, basati rispettivamente su FAT e su Inode. I **puntatori hanno dimensione di 32 bit**, i blocchi hanno dimensione 4KB, entrambi i volumi hanno una parte riservata ai metadata (direttori, FAT e FCB –file control block) oppure Inode, blocchi indice, ecc) e una parte ai blocchi di dato. Si supponga che lo spazio riservato ai blocchi di dato sia lo stesso per F1 e F2. Si sa che la **FAT occupa 150 MB**.

- Siano dati un file “**a.mp4**” (in **F1**) di dimensione **317 MB** e un file “**b.mp4**” (in **F2**) di dimensione **751 MB**.
- G) Quanti blocchi di dato occupa a.mp4?
- H) Quanti indici nella FAT sono utilizzati per a.mp4?

G) Dunque:

- a.mp4 size = 317MB
- Block size = 4KB

Ne consegue che: $317MB/4KB = 79,25K$ blocchi occupati da a.mp4 (cioè 81.152 blocchi, perchè $79,25 \times 1024$).

H) Abbiamo bisogno di tante entries quanti sono i blocchi che occupa a.mp4, dunque 79,25K indici.

Exercise 1:

Siano dati due file system F1 e F2, su due volumi, basati rispettivamente su FAT e su Inode. I **puntatori hanno dimensione di 32 bit**, i blocchi hanno dimensione 4KB, entrambi i volumi hanno una parte riservata ai metadata (direttori, FAT e FCB –file control block) oppure Inode, blocchi indice, ecc) e una parte ai blocchi di dato. Si supponga che lo spazio riservato ai blocchi di dato sia lo stesso per F1 e F2. Si sa che la **FAT occupa 150 MB**.

- Siano dati un file “**a.mp4**” (in **F1**) di dimensione 317 MB e un file “**b.mp4**” (in **F2**) di dimensione 751 MB.
- F) Quanti blocchi di dato e di indice (escluso l’inode) occupa b.mp4? (10 direct blocks, 1 single indirect blocks and 1 double indirect blocks)

F) Dunque:

- b.mp4 size = 751MB
- Block size = 4KB

Ne consegue che: $751MB / 4KB = 187,75K$ blocchi $\Rightarrow 192.256$ blocchi di dati ($187,75 * 1024$).

10 di questi blocchi li indirizziamo tramite **direct blocks**, dunque $192.256 - 10 = 192.246$.

Poi usiamo 1

single indirect block $\rightarrow 192.246 - 1024 = 191.222$.

Poi usiamo 1

double indirect blocks $\rightarrow 191.222 / 1024 = 186,74 = 187$ blocchi di indice di cui abbiamo ancora bisogno.

Dunque in totale abbiamo: $10 + 1024 + 187 = 1221$ blocchi di indice necessari per indirizzare tutti i nostri blocchi di dati.



Double indirect blocks

Facciamo diviso 1024

(*in questo caso abbiamo blocchi da 4KB e puntatori da 4B, dunque $4KB / 4B = 1024$*) perchè nei double indirect blocks abbiamo **un** blocco di indice composto da 1024 entries in totale, ciascuna delle quali è un puntatore ad un altro blocco di indice. Ciascun blocco di indice puntato è composto da 1024 entries, ciascuna delle quali è un puntatore ad un blocco dati, ciascuno di essi composto da 1024 entries (a loro volta, ciascuna entry è 4B, ma di dati questa volta!).

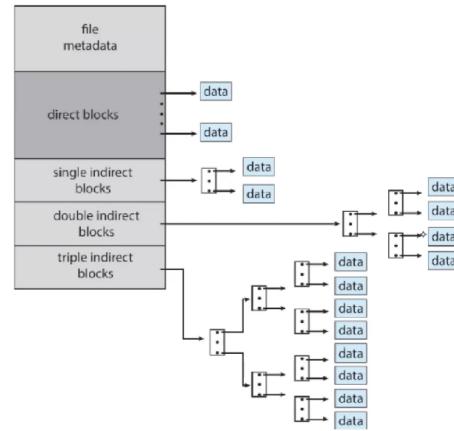
Dunque per capire quanti puntatori usiamo del blocco di indice "a monte", dobbiamo dividere i blocchi di dati che rimangono, 191.222, per 1024 — ovvero andiamo a vedere quanti blocchi di indice "a valle" andiamo a coprire.

Il risultato è 187, e quindi abbiamo bisogno di 187 entries (e cioè puntatori) nel blocco di indice a monte.

Esercizio 2

Exercise 2:

- Sia dato un file system Unix, basato su inode aventi 13 puntatori (10 diretti, 1 indiretto singolo 1 doppio e 1 triplo). I puntatori hanno dimensione di 32 bit e i blocchi hanno dimensione 2KB. Si sa che il file system contiene 1000 file di dimensione media 15MB e che la frammentazione interna totale è di 1MB, si calcoli il massimo numero possibile di file con indice indiretto triplo e con indice indiretto doppio che possono essere presenti nel file system.



Dunque, calcoliamo innanzitutto quante entries hanno i nostri blocchi:

- Puntatore = 32bit = 4B
- Block size = 2KB

Ne consegue che: $2KB/4B = 512$ entries.

Il file system contiene **massimo** (non c'è scritto, ma dovrebbe) 1000 files di dimensione media 15MB. Inoltre, la frammentazione interna **totale** è di 1MB.

Ne consegue che al massimo possiamo memorizzare:

$(1000 * 15MB) + 1MB = 15001MB \rightarrow \text{total files size} — \text{cioè quanto occupano in totale tutti questi 1000 files.}$

Di quanti blocchi abbiamo bisogno per memorizzare, in generale, questi dati?

Dunque avremo:

$15001MB/2KB = 7500,5K$ blocchi di dati in totale (7.680.512 blocchi, perchè $7500,5 * 1024$).

L'esercizio ci chiede però di calcolare **il massimo numero di file** che possiamo memorizzare, arrivando ad usare in un caso l'indice indiretto doppio, e nell'altro l'indice indiretto triplo.

Dunque abbiamo:

$$\text{number of files} = \frac{\text{total file size}}{\text{single file size}}$$

Vorremmo massimizzare **number of files**, e quindi dobbiamo minimizzare **single file size**, sapendo che in un caso dobbiamo usare indice indiretto doppio, e nell'altro caso indice indiretto triplo.

Esaminiamo prima il caso di **indice indiretto doppio**.

Per sapere il massimo numero di files che possiamo memorizzare sapendo di usare (per tutti i files) l'indice indiretto doppio, dobbiamo calcolare l'occupazione minima di ciascun file — cioè quanti blocchi di dati ogni file occupa sapendo di arrivare ad usare l'indice indiretto doppio.

Noi vorremmo arrivare ad usare l'indice indiretto doppio, dunque sapendo che abbiamo 10 blocchi di indice diretti, e che il single indirect block è un blocco composto da 512 puntatori a blocchi dati, per arrivare ad usare anche l'indice indiretto doppio abbiamo bisogno almeno di:

$$10 + 512 + 1 = 523 \text{ blocchi di dato}$$

In questo modo, il 523esimo blocco lo dovremo per forza indirizzare tramite double indirect blocks.

In conclusione, il massimo numero di files lo otteniamo:

$$\text{max files number} = \frac{7.680.512}{523} = 14.685$$

In cui 7.680.512 è la quantità di blocchi di dati effettiva che dobbiamo memorizzare, che andiamo a dividere per 523 che è l'occupazione minima (cioè il minimo numero di blocchi) di ciascun file usando indice indiretto doppio.

Nel caso di **indice indiretto triplo**, dovremmo ripetere il calcolo sapendo che, per arrivare ad usare l'indice indiretto triplo, avremmo bisogno di: $10 + 512 + 512^2 + 1 = 262.667$ blocchi di dato.

In tal caso avremo:

$$\text{max files number} = \frac{7.680.512}{262.667} = 29$$

Exercise 2:

- Un blocco indice contiene $2KB/4B = 512$ puntatori.
- La dimensione (netta, indipendente dal tipo di file system) complessiva dei file è $15MB * 1000 = 15000MB (= 15GB$ circa)
- L'occupazione (questa dipende dal file system e tiene conto dei blocchi effettivamente usati), tenendo conto della frammentazione interna, è $15001MB = 15001 * 512$ blocchi.
- Calcolo massimo per N2/N3 (numero file con indice indiretto doppio/triplo) Per calcolare il numero massimo occorre considerare l'occupazione minima.
- Si indicano con MIN_2 e MIN_3 le occupazioni minime dei due tipi di file:
 - $MIN_2 = (10 + 512 + 1)$ blocchi
 - $MIN_3 = (10 + \cancel{512} + 512 + 512^2 + 1)$ blocchi
- $N2 = [15001 * 512 / 523] = 14685$
- $N3 = [15001 * 512 / (523 + 512^2)] = 29$
- N2 supera il numero di file presenti (1000) quindi il numero massimo di file con indice indiretto doppio è limitato a 1000.



Nota: $\frac{15001MB}{2KB} = \frac{15001K}{2} = \frac{15001 * 1024}{2} = 15001 * 512$