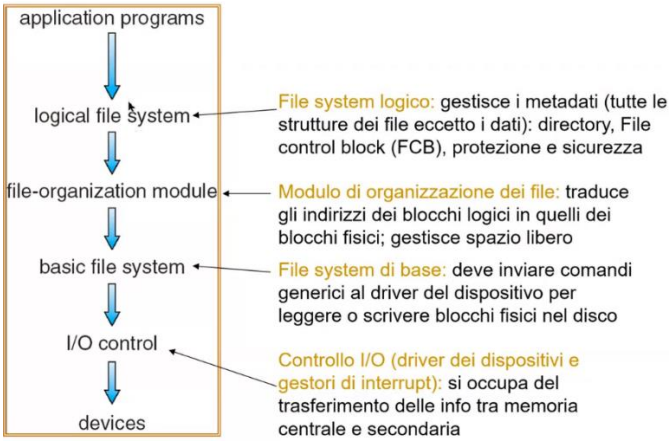
3.0 ORGANIZZAZIONE DEL FILE SYSTEM

Il File System rappresenta la definizione dell'aspetto del sistema agli occhi dell'utente, algoritmi e strutture dati che permettono di far corrispondere il file system logico ai dispositivi fisici. Il File System risiede in un'unità di memorizzazione secondaria (disco):

- Per **leggere** un blocco dal disco, lo si deve portare in memoria principale.
- Per **modificarlo**, lo si deve portare in memoria principale e dopo averlo modificato, lo si riscrive nella stessa posizione del disco.
- Al blocco ci si può accedere con accesso diretto, muovendo la testina di lettura.

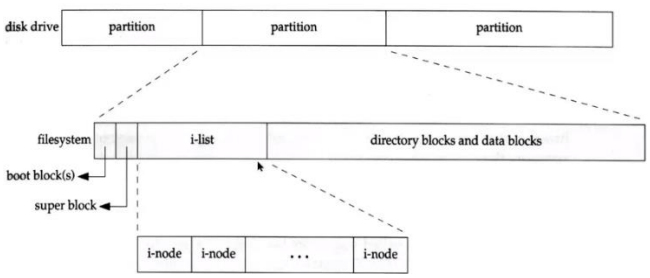
In generale, il file system è organizzato a strati, che segue questa linea:
Supponiamo che il file che stiamo cercando si trovi sulla **devices**, per poter accedere al dispositivo fisico, c'è un **controllo dell'input-output** che si occupa del trasferimento delle informazioni dal disco alla memoria principale e viceversa.
Una volta che il file entra nella memoria principale, il **file system di base** da i comandi al driver, cioè quello che dispone la necessità di leggere o scrivere blocchi fisici nel disco.
Il **modulo di organizzazione dei file** traduce gli indirizzi dei blocchi logici in quelli dei blocchi fisici. Gestisce lo spazio libero, cioè, quando si vuole leggere il contenuto di un file (il quarto byte precisamente), allora il modulo di organizzazione dei file si deve porre il problema di capire dove fisicamente si trova il quarto byte.
Tutto questo viene gestito grazie al **File system logico** che gestisce i metadati (tutte le strutture dei file eccetto i dati): directory, File control block (FCB), protezione e sicurezza.



Sul disco, sono presenti le seguenti strutture:

- **Boot control block**: informazioni necessarie per il caricamento del sistema operativo (*UFS – boot block, NTFS – partition boot sector*).
- **Volume control block**: contiene dettagli del volume, numero di blocchi per partizione, taglia dei blocchi, blocchi liberi … (*UFS, NTFS*).
- **Struttura delle directory**: usata per organizzare i file … (*UFS – nomi file e i -node associato, NTFS – master file table*).
- **File control block (FCB)**: informazioni sul file (*UFS – inode*).

Dopo aver esposto le precedenti informazioni, possiamo ora vedere in più dettaglio quella che è l'organizzazione di un file system in UNIX:
Abbiamo detto che c'è l'i-list, seguito dalla zona effettiva in cui sono immagazzinati i dati. Quello che prima non avevamo visto, è che prima dell'i-list ci sono altre due informazioni che vengono salvate: la prima, il **boot block(s)** in cui vengono caricate le informazioni principali del SO, mentre il **super block** che tiene conto dei dettagli della partizione, del volume che stiamo considerando.

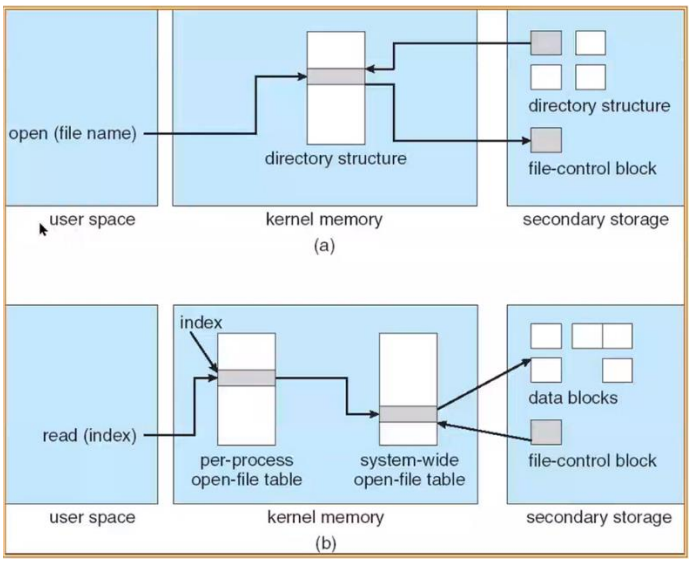


3.1 FILE CONTROL BLOCK

L'insieme degli attributi di un file/directory prende il nome di **File control block** e contiene diverse informazioni, quali i permessi del file, le date del file (creazione, accesso, scrittura), il proprietario, la size e i puntatori ai vari blocchi del file. È quello che in UNIX è detto i-node.

Quando si effettua una richiesta di ***open*** e ***read*** di un file, accade ciò:

Assumiamo che il nostro programma in esecuzione faccia la *open* di un file che nominiamo *file1*. Quando all'interno del campo *file name* della funzione open scriviamo file1, oltre a specificare il file che vogliamo aprire, andiamo anche a specificare qual è la directory all'interno della quale si trova file1. Se non mettiamo la directory, ma scriviamo solo file1, stiamo cercando il suddetto file all'interno della directory corrente, ossia quella in cui è stato salvato il codice.
La prima cosa che il sistema fa quando viene invocata la open, si mette in kernel mode, va a recuperare la directory in questione (si prende il file control block della directory, in cui c'è la lista dei file all'interno della directory, e se lo porta in memoria principale). Nella struttura della directory cerca file1 secondo un criterio. Se non lo trova, la funzione open ritorna un errore, se invece è presente, la directory structure va a recuperare il **file control block** di file1 e lo porta in memoria principale.
Successivamente, all'interno della tabella dei file aperti dal sistema presente in memoria principale, si sarà creata un'entry contenente le informazioni di file1. Verrà creata anche un'entry nella tabella dei file aperti dal processo, a cui verrà assegnato un file descriptor che corrisponde al valore di ritorno della funzione open (rispettivamente le due tabelle nella figura b in memoria principale).



Una volta aperto il file, scorrendo il codice, c'è un'operazione di read. Da questo momento in poi, il riferimento al file non avviene più con il suo nome, bensì attraverso il file descriptor. Per la read a cui si passa in input il file descriptor, si entra in kernel mode, si va nella posizione relativa al file descriptor di file1, all'interno di quest'ultima c'è un campo che punta al riferimento del file nella tabella dei file aperti dal sistema. Si entra nella tabella dei file aperti del sistema al cui interno ci sarà il **File control block** del file di nostro interesse. Tra le cose che ci interessano ci saranno i puntatori ai blocchi di dati veri e propri che ci serviranno per poter leggere il file.

Quando si effettua la open di file1 e si cerca il file all'interno della directory structure, la ricerca può essere fatta in maniera intelligente: per esempio, tenendo una lista ordinata e si effettua una ricerca, oppure organizzare i file contenuti nella directory in una tabella Hash e così via… L'idea è di non cercare sequenzialmente il file altrimenti si perderebbe molto tempo inutilmente.

3.2 METODI DI ALLOCAZIONE

Ci sono 3 tipi di allocazioni standard che ci consentono di effettuare la traduzione da indirizzo logico ad indirizzo fisico: **contigua, linkata e indicizzata**.

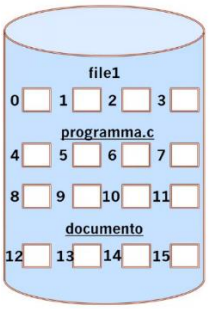
3.2.1 ALLOCAZIONE CONTIGUA:

Ogni file occupa un certo numeri di blocchi contigui (uno dietro l'altro) sul disco. È un'allocazione facile, in quanto una volta che sappiamo dove si trova il primo blocco di un certo file, so facilmente individuare per esempio qual è il quarto blocco del suddetto file. L'accesso può essere di due tipi:

- **Accesso sequenziale**: se per esempio si vuole accedere al decimo blocco di un certo file, quello che si fa è che si legge prima il primo blocco, poi il secondo, …, fino ad arrivare al decimo.
- **Accesso diretto**: se si vuole accedere al decimo blocco di un certo file, l'accesso è diretto, senza passare per i primi 9.

Nel **File Control Block** di **allocazione contigua**, le informazioni che ci interessano sono: **blocco iniziale, lunghezza del file in blocchi**.

Supponiamo di avere a disposizione il disco nell'immagine, in cui sono allocati 3 file posizionati in una directory: file1, programma.c e documento.



File	Blocco iniziale	Lunghezza (in blocchi)
file1	0	4
programma.c	4	8
documento	12	4

Il **File control block** della directory ci dà le informazioni riguardo ai blocchi dei file (sappiamo che il file1 ha come blocco iniziale 0 e la lunghezza è di 4 blocchi), inoltre abbiamo a disposizione un singolo puntatore che punta al blocco iniziale. Naturalmente questo accade perché, sapendo che l'allocazione è contigua, non si ha la necessità di dover avere puntatori a tutti i blocchi di un file, ma basta avere solo quello che punta al primo blocco.

Immaginiamo di esaminare programma.c, sappiamo che comincia al blocco 4 e la lunghezza del file è di 8 blocchi. Se vogliamo accedere al terzo byte del file, tenendo conto che parliamo di allocazione contigua, come facciamo a sapere qual è il blocco che contiene al suo interno il byte 3 del file? La prima cosa che dobbiamo sapere è quanto è grande un blocco (per convenzione supponiamo che ogni blocco è grande 512 byte).

Il calcolo che dobbiamo fare è il seguente:

LA/512 dove LA indica il **Logical Address**, ossia il byte che vogliamo cercare (nel nostro esempio quattro). Quando effettuiamo la divisione, avremo a disposizione un quoziente ed un resto.

Il **quoziente** ci dirà qual è il blocco che contiene il byte di interesse.

Il **resto** ci dirà lo spiazzamento all'interno del blocco, all'interno del quale si trova il byte che stiamo cercando.

ESEMPIO:

La numerazione dei byte dei file parte da 0 e la numerazione dei byte all'interno di ogni blocco parte da 0.

3/512= 0 con resto 3 → significa che il byte 3 è presente nel primo blocco e all'interno del blocco il byte che cerchiamo sta nella posizione 3 (che corrisponde al quarto elemento del vettore),

520/512= 1 con resto 8 → significa che il byte 520 è presente nel secondo blocco (tenendo conto dell'immagine precedente devo effettuare 4+1= 5 blocco, in quanto programma.c parte dal quarto blocco) e all'interno del blocco il byte che cerchiamo sta nella posizione 8, di conseguenza, tenendo conto che la numerazione all'interno di ogni blocco comincia da 0, arrivo all'ottavo (che è in posizione 9) e quello è il byte che mi interessa.

Dal punto di vista del disco, tenendo conto che il file programma.c comincia dal blocco 4, allora dobbiamo effettuare il seguente calcolo: 4 + Q che ci indicherà il blocco effettivo all'interno del quale si trova un certo byte che stiamo cercando.

Gli **svantaggi** dell'allocazione contigua sono:

- **Frammentazione esterna**: tenendo conto dell'immagine precedente, supponiamo che si cancellano ed aggiungono file. Quello che può succedere è che si formano dei buchi, per esempio rimangono 2 blocchi contigui liberi da una parte, altri 2 in un'altra parte e un altro in un'altra parte del disco. Se si riceve una richiesta di un file che richiede 5 blocchi, anche se effettivamente abbiamo 5 blocchi liberi, questi non sono contigui, e di conseguenza il file che richiede 5 blocchi non può essere salvato nel disco.
- **La taglia dei file non può crescere**: supponiamo di prendere come riferimento l'immagine precedente e di focalizzarci su *programma.c* che va dal blocco 4 al blocco 11. Se per qualche motivo si vuole continuare a scrivere all'interno di questo file, questa è una operazione che ci viene vietata in quanto il blocco successivo (ossia il 12) è occupato da *documento*.

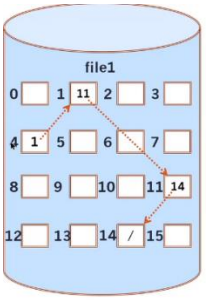
Ci sono alcuni sistemi che usano uno schema di allocazione contigua modificato. Ogniqualvolta non si riesce a far crescere un file in maniera contigua, si cerca una serie di blocchi contigui liberi e si partiziona il file in una serie di blocchi contigui che si trovano sul disco. Naturalmente nel **File Control Block** dobbiamo tenere traccia di dove si ferma un file e da quale blocco riparte. È una tecnica un po' complessa che ha avuto breve vita.

3.2.2 ALLOCAZIONE LINKATA

Con l'allocazione linkata si risolvono i problemi dell'allocazione contigua (frammentazione esterna). Per ogni blocco del disco, si consuma una parte di ogni blocco per contenere un campo puntatore. Di conseguenza, ogni blocco non verrà utilizzato per contenere totalmente dati, ma una piccola parte dovrà essere consumata da un puntatore. Naturalmente il puntatore punterà al blocco successivo presente sul disco.

Nel File Control Block di allocazione linkata, le informazioni che ci interessano sono: **blocco iniziale, blocco finale oppure lunghezza in blocchi (quanti blocchi sono richiesti da un file)**, quest'ultima più usata.

Supponiamo che *MyFileConcatenato* comincia dal blocco numero 4. All'interno del blocco numero 4 ci sono ovviamente le informazioni del file, e ci sarà anche un riferimento al blocco successivo, che in questo caso è il blocco 1. Stesso discorso vale per i blocchi successivi del file. Nel caso in cui, in un secondo momento, si ha la necessità di voler ampliare *MyFileConcatenato*, si cerca un blocco libero sul disco e si va puntare l'ultimo blocco del file a questo nuovo blocco. L'ultimo blocco naturalmente punterà a NULL (/).



File	Blocco iniziale	Blocco finale
MyFileConcatenato	4	14

Vediamo ora come avviene la conversione da indirizzo logico ad indirizzo fisico:

Si tenga conto che per la numerazione dei blocchi, il primo byte, quello che contiene il puntatore non viene considerato. La numerazione parte dal successivo (quindi da 0 a 510).

LA/511 dove LA indica il **Logical Address**, ossia il byte che vogliamo cercare. Quando effettuiamo la divisione, naturalmente avremo a disposizione un quoziente ed un resto. Si osserva che in questo caso il divisore è 511 in quanto stiamo assumendo che il blocco sia di 512 byte di cui 1 byte è occupato dal puntatore al prossimo blocco. Il quoziente ci dirà qual è il blocco che contiene il byte di interesse. Il resto ci dirà lo spiazzamento all'interno del blocco, all'interno del quale si trova il byte che stiamo cercando.

ESEMPIO:

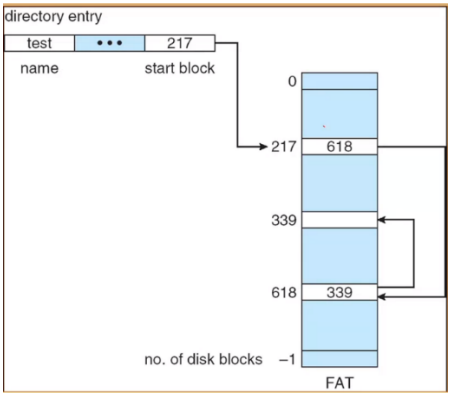
520/511= 1 con resto 9 → ciò significa che il byte 520 è presente nel secondo blocco e all'interno del blocco il byte che stiamo cercando sta alla posizione 9 (10 elemento del vettore non tenendo conto del campo puntatore).

Gli svantaggi dell'allocazione linkata sono:

- **Gestione difficoltosa dei puntatori**: se viene perso un puntatore, non ci si può più riferire al continuo del file.
- **Tempi di accesso**: si è obbligati ad accedere sequenzialmente ad ogni blocco del file, causando continui accessi al disco (in quanto nel File Control Block abbiamo a disposizione solo il riferimento al primo blocco).
- **Consumo di memoria**: per ogni blocco siamo costretti a perdere una quantità definita di byte per il campo puntatore.

3.2.3 TABELLA DI ALLOCAZIONE DEI FILE (FAT)

FAT sta per **F**ile **A**llocation **T**able e funziona nel seguente modo:



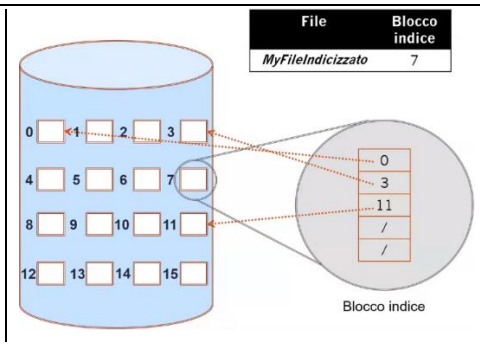
Supponiamo che nel File Control Block del nostro file si trovi semplicemente un numeretto (**start block**) che rappresenta il riferimento all'interno della FAT dentro cui è contenuto il primo blocco del nostro file. La FAT è un array che ha tante entry quanti sono i blocchi utilizzabili del nostro disco. All'interno delle entry dell'array c'è l'indice del successivo blocco di un certo file. Per esempio, tenendo conto del file *test*, nel suo File Control Block all'interno del campo start block c'è il numero 217. Per sapere qual è il blocco successivo, si va all'interno della FAT in corrispondenza dell'elemento con indice 217, si osserva il valore al suo interno che in questo caso è 618, il quale rappresenta quindi il blocco successivo. 618 a sua volta contiene 339 che è il terzo blocco del file. Quando si arriva ad un elemento del vettore che al suo interno contiene il valore NULL allora tale blocco rappresenta l'ultimo di un certo file.

La FAT risiede normalmente sul disco, quando avviene la formattazione di una periferica con modalità FAT, una parte della periferica viene utilizzata per immagazzinare dati e una parte verrà usata per servizio.

Quando viene montato il File System (per esempio quando si mette una penna USB) formattato FAT 32, succede che la tabella FAT viene ricopiata dal disco alla memoria principale. Di conseguenza, in memoria principale abbiamo da subito a disposizione tutte le informazioni sui file contenuti sul disco. Quando per esempio apriamo il file test in didascalia, prendiamo il suo File Control Block e lo portiamo in memoria principale, e se per esempio vogliamo accedere al secondo blocco di questo file, questa operazione viene fatta immediatamente dalla memoria in quanto, come detto in precedenza, abbiamo da subito a disposizione tutta la tabella della FAT in memoria principale. Una volta che sappiamo che il secondo blocco è 618 andremo poi sul disco e lo preleveremo. Abbiamo fatto quindi un solo accesso al disco (se fosse stata l'allocazione linkata avremmo dovuto fare 2 accessi a disco).

Lo **svantaggio** della FAT è dovuto alla grandezza, in quanto già di suo la FAT occupa tanto spazio all'interno del disco e nella memoria principale. Se consideriamo FAT 32, questo 32 rappresenta la lunghezza in bit dell'indirizzo, di conseguenza 4 byte vengono utilizzati per l'indirizzo per individuare ciascuno dei blocchi. Il numero di entry della tabella FAT è 2^{32} e la grandezza di ogni entry della tabella è 4 byte, di conseguenza la grandezza della tabella FAT è $4*2^{32}= 2^{34}$ che sarebbero 16 GB. 16 GB vengono quindi occupati solamente per la FAT.

3.2.4 ALLOCAZIONE INDICIZZATA



In questo tipo di allocazione si prende uno dei blocchi che si assegna ad un file, e all'interno di questo blocco, anziché mettere i dati, lo utilizziamo per mettere gli indici di tutti i blocchi utilizzati per un certo file. Lo possiamo vedere come blocco di servizio.

Possiamo dunque vederlo come raffigurato nell'immagine. Supponiamo di avere un file chiamato *MyFileIndicizzato*. All'interno del File Control Block abbiamo a disposizione il **blocco indice** ossia il riferimento al blocco che contiene al suo interno tutti gli indici del file. Il blocco 7 lo possiamo quindi vedere come un array contenente in modo ordinato tutti i blocchi del file.

Uno svantaggio di questo tipo di allocazione è che per ogni file, dobbiamo perdere un blocco, che dovrà contenere gli indici dei blocchi che costituiscono un file. Se ad esempio un file è costituito da un solo blocco, dovremo avere comunque a disposizione un ulteriore blocco indice.

Vediamo come avviene il mapping da indirizzo logico ad indirizzo fisico:

LA/512 dove LA indica il **Logical Address**, ossia il byte che vogliamo cercare (nel nostro esempio quattro). Quando effettuiamo la divisione, naturalmente avremo a disposizione un quoziente ed un resto. Il quoziente ci dirà qual è il blocco che contiene il byte di interesse.

Il resto ci dirà lo spiazzamento all'interno del blocco, all'interno del quale si trova il byte che stiamo cercando.

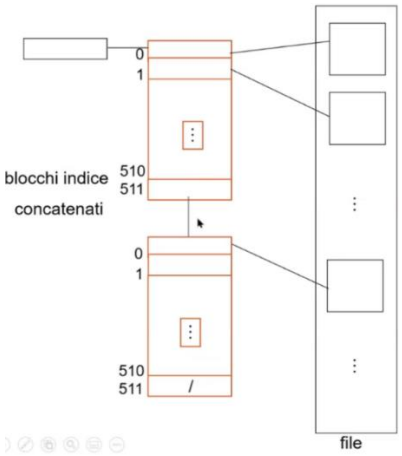
ESEMPIO:

La numerazione dei byte dei file parte da 0 e la numerazione dei byte all'interno di ogni blocco parte da 0

520/512= 1 con resto 8 → significa che il byte 520 è presente nel secondo blocco (tenendo conto dell'immagine precedente devo entrare nel blocco con indice 3) e all'interno del blocco il byte che cerchiamo sta nella posizione 8, di conseguenza, tenendo conto che la numerazione all'interno di ogni blocco comincia da 0, arrivo all'ottavo (che è in posizione 9) e quello è il byte che mi interessa. Il blocco indice lo si porta una sola volta dal disco alla memoria principale.

Un altro **svantaggio** dell'allocazione indicizzata si ha nel momento in cui il file è molto grande, infatti se è molto grande lo spazio contenuto all'interno di un blocco indice non è sufficiente per immagazzinare tutti gli indici dei blocchi che costituiscono un file. Si rimedia a tutto ciò con l'allocazione indicizzata a schema concatenato.

ALLOCAZIONE INDICIZZATA – SCHEMA CONCATENATO:



Supponiamo dunque che un singolo blocco indice non sia sufficiente ad immagazzinare tutti gli indici dei blocchi che costituiscono un file. Quello che si fa in questo caso, è di prendere l'ultimo indice del primo blocco indice e, anziché contenere al suo interno un riferimento ad un blocco del file sul disco, esso conterrà un puntatore ad un ulteriore blocco indice. Se il secondo blocco indice è sufficiente a contenere tutti gli indici dei blocchi del file sul disco, allora l'ultimo indice di tale blocco verrà contrassegnato con "/" che indica un valore NULL. Per essere ancora più chiari, se il primo blocco indice si chiama *a* e il secondo blocco indice si chiama *b* allora all'interno dell'indice 511 del blocco *a*, ci sarà scritto *b* che rappresenta il puntatore al blocco *b*.

In questa modalità, la conversione da indirizzo logico ad indirizzo fisico risulta essere un po' più complesso, scelto il **LA** di interesse, si effettuano questi calcoli:

LA/(511 x 512), la seguente divisione ci fornirà un quoziente (Q_1) e un resto (R_1). Come divisore consideriamo (511 x 512) perché ogni blocco indice mette a disposizione 511 entry contenenti riferimenti a blocchi del file (in quanto 1 indice è perso per il campo puntatore al successivo blocco indice), mentre ogni blocco contenente dati di file contiene al suo interno 512 byte → il numero di byte che posso individuare attraverso un unico blocco indice sono proprio (511 x 512).

Preso Q_1, bisogna addizionargli 1 per individuare qual è il blocco indice che dobbiamo considerare.

R_1 viene invece usato per un ulteriore calcolo:

R_1/512, la seguente divisione ci fornirà un ulteriore quoziente (Q_2) e un resto (R_2).

Q_2 rappresenta l'indice da considerare nel blocco della tabella indice individuato in precedenza.

R_2 rappresenta l'indice nel blocco del file.

ESEMPIO1:

Scelto LA=200, si effettuano i seguenti calcoli:

200/(511*512), Q_1=0, R_1=200.

Questo significa che dobbiamo considerare il primo blocco indice (in quanto Q_1=0 + 1= 1). Adesso sfruttiamo R_1:

200/512, Q_2=0, R_2=200.

Questo significa che all'interno del primo blocco indice individuato mediante Q_1 dobbiamo considerare il primo indice (in quanto Q_2=0) e all'interno del blocco del file considerare il 200 byte contenuto in esso.

ESEMPIO2:

Scelto LA=300000, si effettuano i seguenti calcoli:

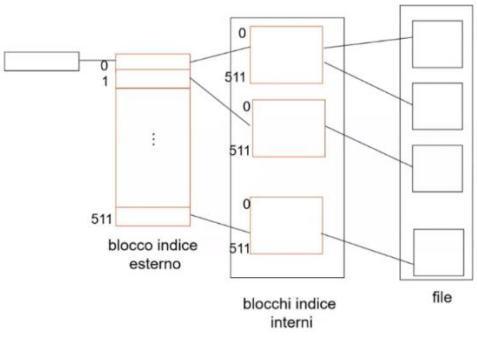
300000/(511*512), Q_1=1, R_1=38368

Questo significa che dobbiamo considerare il secondo blocco indice (in quanto Q_1= 1 + 1= 2). Adesso sfruttiamo R_1:

38368/512, Q_2= 74 , R_2=480

Questo significa che all'interno del secondo blocco indice individuato mediante Q_1 dobbiamo considerare l'indice 74 (in quanto Q_2=74) e all'interno del blocco del file considerare il 480 byte contenuto in esso.

ALLOCAZIONE INDICIZZATA A 2 LIVELLI:



In questo caso c'è un blocco indice esterno e dei blocchi indici interni. Il blocco indice esterno contiene 512 entry ad ulteriori blocchi indici (interni). Ogni blocco indice interno avrà 512 entry in cui ogni indice conterrà un riferimento al blocco del file contenuto nel disco.

In questo caso, il numero di blocchi massimo del file sono 512 x 512, mentre il numero di byte massimi del file sono 512 x 512 x 512.

Vediamo come avviene la conversione da indirizzo logico ad indirizzo fisico:

Scelto il **LA** di interesse si effettuano i seguenti calcoli:

LA/(512 x 512), la seguente divisione ci fornirà un quoziente (Q_1) e un resto (R_1).

Q_1 ci indica quale indice all'interno del blocco indice esterno dobbiamo considerare.

Preso Q_1, bisogna addizionargli 1 per individuare qual è il blocco indice che dobbiamo considerare.

R_1 viene invece usato per un ulteriore calcolo:

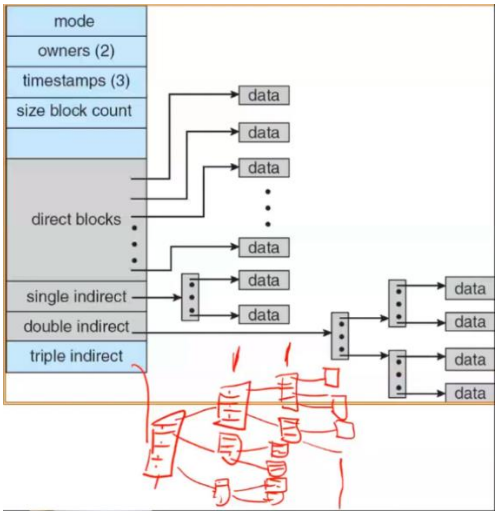
R_1/512, la seguente divisione ci fornirà un ulteriore quoziente (Q_2) e un resto (R_2).

Q_2 rappresenta l'indice da considerare nel blocco indice interno individuato in precedenza.

R_2 rappresenta l'indice nel blocco del file.

3.2.5 SCHEMA COMBINATO: UNIX (4K BYTE PER BLOCCO)

UNIX propone un mix di tutte le allocazioni che abbiamo analizzato precedentemente. Ciò viene fatto per massimizzare la grandezza del file da gestire e per garantire sicurezza e velocità di accesso.



Come mostrato nell'immagine, la parte blu costituisce il FCB di UNIX, cioè l'i-node, che contiene varie informazioni del file. Successivamente, la parte grigia, contiene le informazioni per l'accesso al file. In questa zona ci sono 15 puntatori gestiti nel seguente modo:

- 12 puntatori sono diretti a blocchi (si può vedere come un'allocazione indicizzata).
- 3 puntatori a blocchi indiretti:
  - Il primo punta ad un blocco indiretto.
  - Il secondo punta ad un blocco indiretto doppio (2 strati di blocchi indice).
  - Il terzo punta ad un blocco indiretto triplo (3 strati di blocchi indice).

Assumiamo di avere a disposizione un file F, il cui numero di blocchi è minore o uguale a 12. In questo caso l'accesso ai dati è velocissimo, in quanto vengono sfruttati solo i blocchi contenuti nel ***direct blocks***.

Supponiamo che il file F abbia un numero di blocchi maggiore o uguale a 13. In questo caso ad una parte del file si accederà in maniera diretta, mentre l'altra parte attraverso un accesso a disco utilizzando ***single indirect.***

Sostanzialmente più è grande il file, più tempo ci vorrà ad accedere ad una parte del file.

Questa organizzazione consente di immagazzinare file veramente molto grandi.

Effettuiamo alcuni conti:

Sappiamo che la size di un blocco (data nell'immagine) sia 4kb= 4 x 2^{10}= 2^{12} byte. Adesso bisogna individuare quanto ci costa un campo puntatore all'interno di un blocco indice: tenendo conto che la size di un blocco è 2^{12} e che supponiamo la size di un puntatore sia 4 byte. Allora il numero di puntatori che possono essere messi all'interno di un blocco indice è 2^{12} b / 2^{2} b= 2^{10}.

La size di un file con questo schema può essere al più grande:

$|F| \leq 12 \times \text{size blocco} + 2^{10} \times \text{size blocco} + 2^{10} \times 2^{10} \times \text{size blocco} + 2^{10} \times 2^{10} \times 2^{10} \times \text{size blocco} =$

$\leq 4kb(12 + 2^{10} + 2^{20} + 2^{30})$

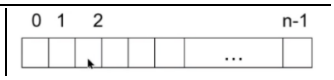
3.3 GESTIONE DELLO SPAZIO LIBERO

Abbiamo visto come poter allocare un file sul disco in vari modi. Nel caso in cui ci sia la necessità di dover far crescere un file c'è bisogno di andare ad individuare un nuovo blocco libero, a seconda di viene gestita l'allocazione.

Qualunque sia l'allocazione utilizzata bisogna sempre avere delle informazioni sui blocchi liberi. Sapere dove sono per poterli rintracciare. Ci sono diverse tecniche per la gestione dello spazio libero:

3.3.1 GESTIONE DELLO SPAZIO LIBERO – VETTORE DI BIT

Abbiamo un vettore con n bit numerati da 0 a n-1. Se all'interno di un indice del vettore c'è uno 0 allora significa che il blocco[i]-esimo è occupato. Se invece c'è un 1, allora tale blocco è libero.



Questo vettore di bit richiede uno spazio extra: ad esempio si supponga che la dimensione di un blocco sia di 2^{12} byte, mentre la dimensione del disco è 2^{30} byte (1 gigabyte). Da queste due informazioni possiamo ricavare n, ossia dei blocchi che possibile inserire che è uguale a $2^{30}/2^{12}$= 2^{18}. Il nostro vettore di bit, di conseguenza, avrà 2^{18} bit che sono occupati solamente per la gestione dei blocchi liberi. Questo rappresenta uno svantaggio.

3.3.2 GESTIONE DELLO SPAZIO LIBERO – LISTA DEI BLOCCHI LIBERI

Lo spazio libero viene gestito mediante una lista a puntatore in cui i blocchi liberi vengono puntati uno dopo l'altro.

Si conserva il puntatore al primo blocco (e solo a questo) in una locazione speciale del disco che viene caricato in memoria quando si vuole accedere ad un blocco libero. In questo modo non c'è spreco di spazio.

Nel caso in cui si richiede un blocco libero, quello che si fa è prendere il blocco 2 in quanto è il primo libero, si porta in memoria principale, si legge qual è il successivo blocco libero (in questo caso il 3), si scrive il 3 nella locazione speciale contenuta all'interno della memoria, in modo tale che da questo momento il blocco 2 è libero e può essere usato e il blocco 3 diventa primo blocco libero della lista dei blocchi liberi.

Altri metodi usati per la gestione dello spazio libero sono:

- **Raggruppamento**: si memorizzano in un blocco gli indici di n blocchi: di questi i primi n-1 sono realmente liberi mentre l'ultimo contiene gli indirizzi di altri n blocchi, e così via. In questo modo si permette di trovare rapidamente molti blocchi liberi.
- **Conteggio**: spesso più blocchi contigui possono essere rilasciati. Quindi ogni elemento della lista dello spazio libero è formato da un indirizzo ed un contatore: l'indirizzo punta al primo dei blocchi liberi mentre il contatore dice quanti blocchi contigui ci sono.