FILE SYSTEM

E' un'astrazione, permette di gestire bene l'organizzazione e memorizzazione dei file in modo non volatile su disco (certe volte anche grandi quantità di dati) inoltre basa anche alla condivisione dei file per i processi

In sintesi: è un sistema di ordinazione su supporto di memoria e gestisce: organizzazione, condivisione, accesso

Un file system è caratterizzato da dettagli di gestione e implementazione:

Dettagli:

- Nomenclatura (come nomina i file)
- tipi di file (estensione) su Unix trovano i magic byte (2 byte iniziali che identificano il tipo di file)
- tipi di accesso (casuale, sequenziale)
- metadati (attributi dei file)
- operazioni sui file (syscall)
  - Accesso condiviso ai file (ci sono i lock)
    - Shared: più processi accedono al file (se devono solo leggere)
    - Exclusive: un solo processo vi accede (vuole scrivere)

Monitory vs Advisory

↓ Meccanismi lock obbligatori

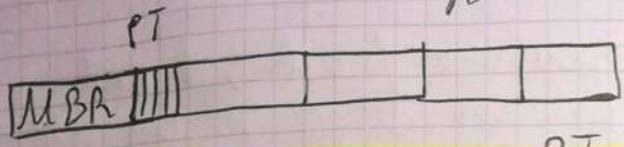
↓ Meccanismi lock suggeriti (responsabilità del programmatore)

Abbiamo strutture dati per la gestione dei file (es: tabella file aperti)

Ogni struttura è o globale o per processo

Struttura di un file system

Standard storico: prevede il Master Boot Record, suddivide in partizioni (ognuna poi partizionata a sua volta)



La prima partizione contiene MBR che boota all'avvio della macchina e PT: una struttura dati molto semplice per le partizioni.

In ogni partizione troviamo blocco di boot e superblocco. Il blocco di boot potrebbe non avviare il SO nella partizione (sotto direttiva di MBR) Se non è presente un SO il blocco di boot diventa inutile ma è comunque presente

Il superblocco (sempre il 2°) contiene meta-info (es: tipo di file system usato ecc)

Un attributo particolare è il cluster: unità minima allocabile per un file

Il SO potrebbe decidere di non usare come cluster 1 blocco ma ad esempio 4 blocchi

A parte questi due primi aspetti poi il resto della partizione può essere gestito in altri modi

Elementi ricorrenti nella partizione:
- root dir → dir principale per rintracciare i file
- files and directories
- i-nodes: tengono traccia delle info dei file

Partizione

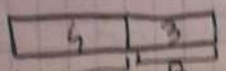
Boot block	Super block	FREE SPACE mgmnt	i-nodes	Root dir	files & dirs

Layout moderno: GPT definito da standard EFI
Permette di partizionare qualunque disco in più parti

Implementazione dei file

NB: i problemi di allocazione RAM sono gli stessi che su Disco.

- **Allocazione contigua**: i blocchi del file hanno mettere tutti contigui



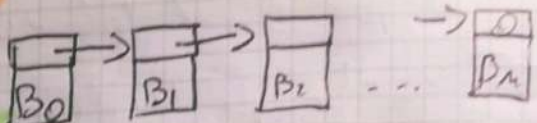
... Vantaggi: gestione facile, il ritardo nel passare da un blocco a un altro vengono meno (sono contigui)

L'allocazione contigua sarebbe l'ideale ma non è fattibile sempre (file ingrossa o si restringe)

La tendenza è quella di allocare contiguamente finché si può per poi passare a tecniche di allocazione non contigua

- **Allocazione con liste collegate**

File F divisi in modi (blocchi/cluster)



Vantaggio: si bypassano i problemi relativi alla allocazione contigua (frammentazione esterna)
La frammentazione interna continua ad esistere (nell'ultimo)
Il fatto di usare dei byte per il pointer next può creare disallineamenti (che influenzano le prestazioni)

I disallineamenti sono dovuti al fatto che i dati veri e propri magari sono più corti una dimensione potenza del 2

L'accesso sequenziale funziona abbastanza bene ma risulta difficile e inefficiente l'accesso casuale.

Modifichiamo questa tecnica per evitare un accesso casuale lento

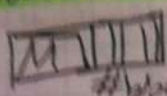
- Allocazione con liste collegate su una tabella di allocazione dei file

FAT (file allocation table): è una tabella/struttura del file system che dello stile Unix fornisce una tabella che permette l'accesso casuale
Di fatto avevamo una lista per ~~ogni~~ file ora abbiamo una tabella FAT e basta
La voce del blocco 5 ha il numero del next blocco per la lettura del file (-1 se è l'ultimo)
Nella dir c'è anche l'informazione per capire se è il blocco iniziale o no
Lo standard prevede due tabelle per motivi di resistenza ai guasti (le tabelle sono ridondate)
Potrei mettere un valore fittizio (non -1) per capire se un blocco non è coinvolto in nessun file
La FAT sarebbe perfetta per l'accesso casuale (sarebbe preferito portare la FAT per intero in RAM)
Portare la FAT su RAM è costoso se no a che fare con dischi enormi. Si risolve molto semplicemente mappando/impaginando la FAT

- Allocazione con i-nodes

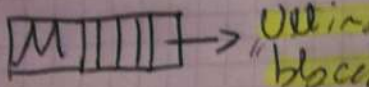
i-node: struttura dati (piccola) che contiene info sull'oggetto/file
ID del file/oggetto. Contiene metadati fondamentali per la gestione del file
L'i-node riguarda un singolo file, lo portiamo su RAM solo se il file sta su RAM
Sulla RAM stanno solo gli i-nodes che servono
L'i-node number identifica l'i-node
L'i-node contiene tutti i metadati del file (tranne il nome che è nella dir)

La directory si immagina come una tabella dove
Ogni record è • i-number • name
Nel modello PAT la voce è più grossa e già grossa
Vi sono tutti i metadati del file
L'i-node deve tener traccia in qualche modo
dei vari pezzi del file sul disco.
L'i-node è formato da metadati e voci dette
numeri di blocchi

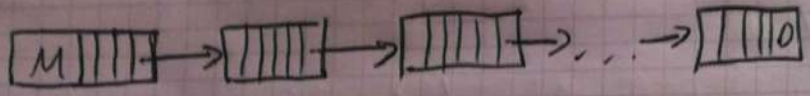


Ogni voce tiene info su un blocco.
Se ho 10 voci, posso gestire con un i-node
un file di max 10 blocchi
Se il file è grosso? Estendo l'i-node

Strategie:
• Uso l'ultima voce già puntare a un blocco nel
disco che contiene delle voci che puntano ad
altri blocchi (abbiamo fatto una 'estensione')
L'estensione può avvenire più volte
L'ultima voce è detta Address of block of pointers



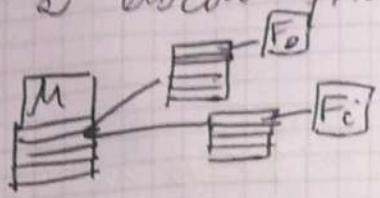
Così estendiamo la lista iniziale se necessario
Si crea la lista collegata



Accesso sequenziale ottimo
Accesso casuale inefficiente

Potrei in realtà fare una struttura ad albero
detta multilivello
Ogni voce dell'i-node punta a una lista di voci
che puntano a blocchi file
È un albero dove la radice è l'i-node e le voci

puntano ai suoi figli (le sequenze di voci che puntano ai blocchi fisici)



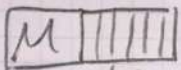
L'albero se il file è piccolo non è completo

L'accesso diretto diventa più facile (prima era impossibile, se volevo Fi dovevo per forza scorrere la lista)

Problema: composta 1 lettura extra nel disco
La libera file è quasi identica a quella della tabella delle pagine.

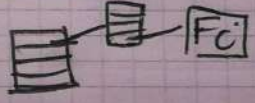
Potrei prevedere anche più livelli (più overhead)
Problema: i file piccoli verrebbero gestiti male con questa tecnica (troppo spreco)
Che approccio si usa? IBRIDO

i-node Unix



Uso le prime 10 voci come blocchi diretti (puntano a blocchi fisici)
Se devo gestire file che usano più di 10 blocchi le ultime 3 voci mi aiutano.
La terzultima punta a

estensione della lista (ciascuna di queste fa riferimento a blocco)
Ogni voce dell'estensione punta a blocchi fisici.
Diciamo che la terzultima permette di gestire file di dim "media"
Se devo accedere a $F_{10-1033}$: overhead di 1
Se non basta ancora uso la penultima voce che punta a un albero a 2 livelli



Overhead (per accedere qui): 2
Già il file gestibile è abbastanza grosso
L'ultima voce (se ci serve) punta a un albero a 3 livelli

Puoi gestire file enormi. Overhead: 3

Supponendo che un blocco / cluster sia 4KB

File "piccolo": 10 × 4 KB

File "medio": (10+1024) × 4KB

File "mediogrande": (10+1024+1024²) × 4KB

File "grande": (10+1024+1024²+1024³) × 4KB

L'overhead in realtà si può gestire con una qualche ottimizzazione.

(Esempio: overhead iniziale e/o ogni I/O del contenuto vicino in RAM così se devo usare quel contenuto ho evitato di fare l'overhead più volte)