

Operazioni collegate fra loro

Quando creo un file con, ad esempio

```
$ echo "ciao" > /tmp/foo
```

Il filesystem viene modificato in tre punti:

0. creo un nuovo inode
1. alloco un blocco di dati
2. modifico una directory

Cosa succede se manca la tensione ed eseguo solo una o due di queste operazioni?

⇒ il filesystem può trovarsi in uno stato *inconsistente*

1

Dati e metadati

Un filesystem contiene

- *dati* (il contenuto dei file), e
- *metadati* (superblocco, inode, bitmap, directory, FAT,...)

Il filesystem è *inconsistente* se c'è un errore nei metadati

I filesystem sono corredati di un programma per correggere le inconsistenze

- MS-DOS, Win95: scandisk.exe
- Unix: fsck

2

File-system check

Il filesystem ext2 ha un "clean bit"

Quando è montato il FS viene marcato "not clean"

Quando è smontato il FS viene marcato "clean"

⇒ se all'avvio i filesystem sono marcati "not clean" vuol dire che il sistema è caduto (spento in maniera non regolare)

In tal caso fsck(8) viene eseguito automaticamente

Richiede vari minuti, di più se il filesystem è grande

3

Funzionamento di fsck(8) (i)

a. check blocks

alloca una tabella di contatori $c[\text{NR_BLOCKS}]$

per ogni inode in uso

per ogni blocco i appartenente al file

incrementa $c[i]$

Se a un certo punto mi ritrovo con $c[k] > 1$ vuole dire che il blocco k appartiene a più di un file

⇒ lo riparo duplicando il blocco

Scorro la bitmap dei blocchi liberi

- se un blocco j risulta libero ma ha $c[j] > 0$ allora risulta sia libero che occupato

⇒ lo riparo segnando nella bitmap che il blocco è occupato

- se un blocco risulta occupato ma ha $c[j] = 0$ allora è un blocco "perduto"

⇒ lo riparo segnando nella bitmap che il blocco è libero

4

Funzionamento di fsck(8) (ii)

b. check files

alloca una tabella di contatori $c[\text{NR_INODES}]$

partendo da / discende ricorsivamente tutte le directory

per ogni directory entry incrementa $c[i]$ dove i è l'inode corrispondente

alla fine, per ogni k confronta $c[k]$ con il link count nell'inode k

- se $c[k] \neq$ link count allora aggiorna il link count

5

Problemi con fsck

- è lento (e diventa più lento all'aumentare della dimensione dei dischi)
- "corregge gli errori" ma quando interviene i dati dell'utente sono già persi

6

Transazioni

Consideriamo l'esempio di una banca con le seguenti operazioni:

- `deposit(name, amount)`
deposita una somma sul conto "name"
- `withdraw(name, amount)`
preleva una somma
- `get_balance(name) → amount`
restituisce il saldo del conto "name"
- `branch_total() → total`
restituisce la somma di tutti i depositi

7

Un semplice esempio di transazione

Il cliente A fa un bonifico sul conto B

0. `withdraw(A, 100)`
1. `deposit(B, 100)`

Cosa succede se il database cade dopo avere fatto 0 ma prima di fare 1?

8

Primo requisito di una transazione: l'atomicità

all-or-nothing

Una transazione o viene completata con successo o non ha alcun effetto

- failure atomicity: gli effetti sono atomici anche se il server cade
- durability: se una transazione ha successo tutti i suoi effetti sono salvati su memoria permanente

9

Un'altro esempio

Due transazioni eseguite contemporaneamente

Transaction T0	Transaction T1
withdraw(A, 100) deposit(B, 100)	branch_total()
balance = read(A); write(A, balance - 100);	total = 0;
balance = read(B); write(B, balance + 100);	total += read(A); total += read(B); total += read(C); :

La somma dei saldi non è corretta!

10

Secondo requisito delle transazioni: isolamento

isolation

Ciascuna transazione deve essere eseguita senza interferenza

11

Un'altro esempio ancora

Il database della banca conserva in una variabile globale il valore di branch_total

```
Transazione corretta  
deposit(A, 100)  
balance = read(A);  
write(A, balance + 100);  
branch_total += 100;
```

Se il programmatore dimenticasse di aggiornare branch_total il DB risulterebbe *inconsistente*

12

Terza caratteristica delle transazioni: consistenza

consistency

Mentre le prime due caratteristiche (all-or-nothing, isolation) sono implementate dal DBMS, la consistenza delle transazioni è responsabilità del programmatore applicativo

13

ACID

- Atomicity: una transazione deve essere all-or-nothing
- Consistency: una transazione porta il sistema da uno stato consistente a un'altro stato consistente
- Isolation: una transazione *in corso* non deve influenzare altre transazioni
- Durability: gli effetti di una transazione sono permanenti

14

Come implementare le transazioni?

Sappiamo implementare l'atomicità con i *mutex*

Due problemi:

- se forziamo la serializzazione delle transazioni, perdiamo in performance (potremmo eseguire le transazioni in parallelo)
- il mutex non garantisce l'atomicità se il sistema cade!

15

Fail-safe durability

0. Tutti i blocchi da modificare vengono *bloccati* (locked) nella buffer cache (cioè non possono essere copiati sul disco)

1. La transazione viene *scritta* nel *journal*

2. I blocchi vengono sbloccati

3. Quando l'ultimo dei blocchi è stato scritto con successo sul disco, la transazione è marcata *completa* nel journal

all'accensione il sistema esamina il journal

- se ci sono transazioni aperte ma non completate, queste vengono *rieseguite* (replayed)

16

Esempio di transazione

Vogliamo creare un nuovo file; assumiamo di dover modificare due blocchi

```
|         |         |         | scrivi scrivi | marca |
| alloca | aggiungi nome | scrivi transazione | blocco blocco | completata |
| i-node | alla direct. | sul journal       | 33 41 | la transaz. |
A-----B-----C-----D-----E-----F
blocco   blocco
 41      33
```

- se il sistema cade prima di "D", la transazione non è mai avvenuta
- se il sistema cade fra "D" e "F", la transazione viene rieseguita
- dopo "F", la transazione è permanente

17

Journaling

Il file di journal contiene una serie di *entrate* che possono essere:

- `begin_transaction T`
- `commit_transaction T`
- `abort_transaction T`
- `modify resource old-value new-value`

L'entry *modify* contiene abbastanza informazioni per poterla *disfare*, anche nel caso in cui non so se l'operazione sia stata fatta oppure no

18

Esempio di journal

```
x := 0
y := 0
BEGIN_TRANS
x := x + 1
y := y + 2
x := y * y
END_TRANS
```

```
journal:
-----
begin_trans
modify x 0 1
modify y 0 2
modify x 1 4
end_trans
```

19

Torniamo ai filesystem

Quando creo un file con, ad esempio

```
$ echo "ciao" > /tmp/foo
```

Il filesystem viene modificato in tre punti:

0. creo un nuovo inode
1. alloco un blocco di dati
2. modifico una directory

questa operazione deve essere eseguita come *transazione*

esistono filesystem costruiti intorno al concetto di *journal*

il vantaggio: elimino fsck; il recovery prende pochi secondi anche per terabyte di dati

altro vantaggio: riduco il rischio di perdere dati

20

La scelta più conservatrice

ext3 è l'evoluzione di ext2

ext3 = ext2 + journal

facile da usare

sicuro

un volume ext3 può essere montato come ext2

tratta in maniera transazionale dati e metadati

21

Vivere sul filo del rasoio dell'innovazione

reiserfs è un nuovo filesystem creato da Hans Reiser

- journaling
- ottima performance
- ottima small-file performance
 - dati e inode sono conservati nello stesso blocco
 - tail-packing: più piccoli file (o code di file) sono compresse nello stesso blocco
- può contenere il 6% di dati in più rispetto a ext2/ext3

22

Ancora su reiserfs

Un filesystem estendibile

- posso definire il *mio* formato per directory, il *mio* formato per i file

(es. posso sviappare un estensione per trattare i file .zip come directory in maniera trasparente)

23

Application-level transactions

I journaled filesystem trattano ciascuna operazione come una transazione: dati e metadati vengono aggiornati atomicamente

Ma il filesystem non ha modo di sapere quale sequenza di operazioni sul disco costituisce una transazione *per l'applicazione*

La soluzione tradizionale: usare un DBMS

reiserfs fornirà un API per programmare il filesystem in maniera transazionale

(altro esempio del confine labile fra DBMS e FS)

24

NTFS

NTFS: un *nuovo* filesystem per Windows, incompatibile con MS-DOS e Win9x

Un file NTFS *non* è una semplice sequenza di byte

Un file consiste di vari attributi, ciascuno è una sequenza di byte

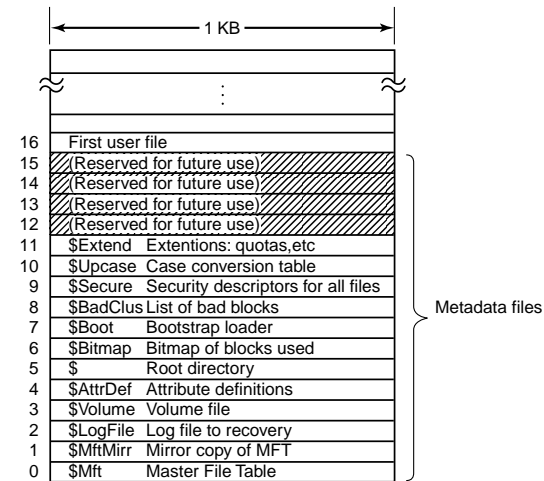
L'attributo senza nome è il "contenuto" tradizionale del file (con semantica Unix)

Applicazioni:

- conservare una thumbnail di una immagine
- conservare la resource fork di un file Macintosh
- ... a disposizione del programmatore applicativo

25

The NTFS master file table (MFT)



26

The NTFS master file table (MFT)

Ogni record nella MFT ha le funzioni di un i-node

- attributi
- lista dei blocchi
- formato: lista di attributi nome-valore

I primi 16 record puntano a metadata files (il nome inizia per \$)

La MFT è un file anch'essa e può stare ovunque nel filesystem

(L'indirizzo del primo blocco della MFT è nel boot block)

File 0 (\$Mft) si riferisce alla MFT stessa!

File 1 (\$MftMirr) copia della MFT

27

File 2 (\$LogFile) journal per fail-safe atomicity

File 3 (\$Volume) informazioni sul filesystem (cf. superbloc)

File 4 (\$AttrDef) definizione degli attributi per la MFT

File 5 (\$) root directory

File 6 (\$Bitmap) bitmap dei blocchi liberi/occupati

Attributi di un file

- Standard information: flag bits, timestamps, ...
- File name
- Attribute list: punta ad altri record MFT se necessari
- Object ID: identificatore di 64 bit unico per volume
- Data: sequenza di byte; ce ne può essere più d'una

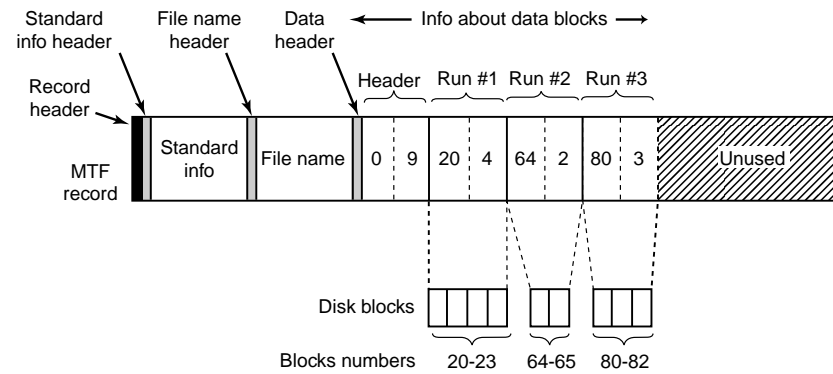
Se un attributo è piccolo → è conservato nella MFT;
altrimenti è un *attributo non residente*

Il nome dell'attributo non residente contiene anche la lista dei blocchi

I file di poche centinaia di byte hanno dati residenti nella MFT

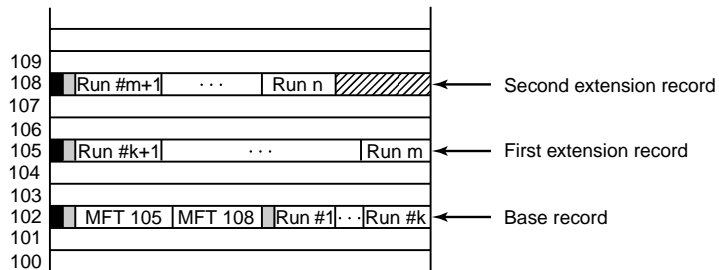
28

Il record MFT per un file di 9 blocchi in 3 serie



29

Il caso di un file con "troppe" serie di blocchi



30

RAID

I processori raddoppiano di velocità ogni 18 mesi (legge di Moore)

I dischi no.

Dunque l'accesso ai dischi diventerà sempre più il fattore limitante

Però i dischi dimezzano di prezzo a una velocità anche maggiore

Possiamo mettere insieme più dischi per parallelizzare l'accesso

⇒ Redundant Array of Inexpensive Disks (RAID)

31

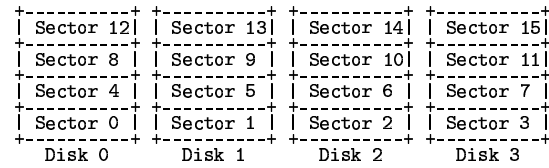
RAID

Due possibili implementazioni:

- in hardware (ho una scatola piena di dischi, con un solo controller)
- in software (il S.O. mi permette di vedere più dischi come se fossero uno solo)

32

RAID level 0

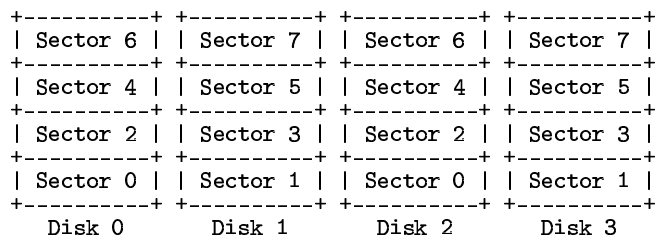


pro: le richieste di I/O sono distribuite su più dischi ⇒ migliore performance

contro: se fallisce uno solo dei dischi, ho perso tutto il filesystem

33

RAID level 1



ogni disco è replicato

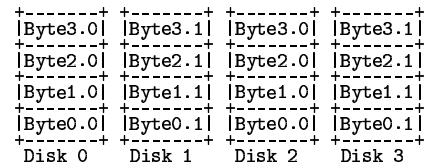
pro: se un disco si rompe, il sistema continua a funzionare

pro: le letture possono raddoppiare velocità

contro: il costo per byte raddoppia

34

Raid level 2



distribuisce i *byte*, non i settori

⇒ ora per leggere 4 byte ci metto un quarto del tempo

pro: massimizza il throughput

contro: nella maggior parte dei casi mi interessa il seek time, non il throughput

poco usata in pratica

35

Raid level 3

È lo stesso del level 2, ma con un disco di *parità*

Es: il bit 0 del disco di parità contiene lo XOR del bit 0 di tutti gli altri dischi

Se **un** disco fallisce posso usare la parità per ricostruire il disco mancante

Abbiamo quasi la stessa affidabilità del livello 1, senza bisogno di raddoppiare i dischi

36

Raid level 4

Sector 8	Sector 9	Sector 10	Parity8-10
Sector 6	Sector 7	Sector 8	Parity6-8
Sector 3	Sector 4	Sector 5	Parity3-5
Sector 0	Sector 1	Sector 2	Parity0-2
Disk 0	Disk 1	Disk 2	Disk 3

È come il livello 0 con in più un disco di parità

pro: velocizza la lettura

pro: un solo disco di ridondanza

contro: le scritture devono toccare due dischi, quello dei dati e quello di parità

contro: se un disco cade, per leggere un byte che stava sul disco caduto devo leggere tutti i byte corrispondenti degli altri dischi

⇒ la performance degrada in maniera drammatica (in livello 1 invece no)

37

Raid level 5

Parity8-10	Sector 8	Sector 9	Sector 10
Sector 6	Parity6-8	Sector 7	Sector 8
Sector 3	Sector 4	Parity3-5	Sector 5
Sector 0	Sector 1	Sector 2	Parity0-2
Disk 0	Disk 1	Disk 2	Disk 3

Variante del livello 4: i settori di parità sono distribuiti

Più veloce in scrittura (le scritture possono essere parallelizzate)

Più veloce in lettura (con 4 dischi la velocità è 4x; in level 4 con 4 dischi è solo 3x)

38

RAID in pratica

I livelli più usati sono 1 (mirroring) e 5 (striping)

Il mirroring è più semplice: se ho un guasto l'altro disco è una replica completa del filesystem

Con level 4 o 5 ho maggiore performance, ma in caso di caduta ricostruire le informazioni del disco rovinato richiede di eseguire un programma di recupero

Windows NT/2k e Linux implementano RAID in software

39