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*

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

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

Funzionamento di fsck(8) I

1. check blocks
   - 1.1 alloca una tabella di contatori c[NR_BLOCKS]
     per ogni inode in uso
       per ogni blocco i appartenente al file
         incrementa c[i]
   - 1.2 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
   - 1.3 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

Funzionamento di fsck(8) II

2. check files
   - alloca una tabella di contatori c[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

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

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) $\rightarrow$ amount
  restituisce il saldo del conto "name"
- branch_total() $\rightarrow$ total
  restituisce la somma di tutti i depositi

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?

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

Un'altro esempio

Due transazioni eseguite contemporaneamente

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

	total += read(A);
	total += read(B);
	total += read(C);

balance = read(B); write(B, balance + 100);	⋮

La somma dei saldi non è corretta!

Secondo requisito delle transazioni: isolamento

isolation

Ciascuna transazione deve essere eseguita senza interferenza

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*

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

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

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!

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)

Esempio di transazione

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

```
|        |               |                 | scrivi scrivi | marca       |
| alloca | aggiungi nome | scrivi transaz. | 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

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

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

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

Ancora su reiserfs

Un filesystem estendibile

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

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

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)

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

$\Rightarrow$ Redundant Array of Inexpensive Disks (RAID)

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)

RAID level 0

```
+----------+ +----------+ +----------+ +----------+
| Sector 12| | Sector 13| | Sector 14| | Sector 15|
+----------+ +----------+ +----------+ +----------+
| Sector 8 | | Sector 9 | | Sector 10| | Sector 11|
+----------+ +----------+ +----------+ +----------+
| Sector 4 | | Sector 5 | | Sector 6 | | Sector 7 |
+----------+ +----------+ +----------+ +----------+
| Sector 0 | | Sector 1 | | Sector 2 | | Sector 3 |
+----------+ +----------+ +----------+ +----------+
   Disk 0       Disk 1       Disk 2       Disk 3
```

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

RAID level 1

```
+----------+ +----------+ +----------+ +----------+
| Sector 6 | | Sector 7 | | Sector 6 | | Sector 7 |
+----------+ +----------+ +----------+ +----------+
| Sector 4 | | Sector 5 | | Sector 4 | | Sector 5 |
+----------+ +----------+ +----------+ +----------+
| Sector 2 | | Sector 3 | | Sector 2 | | Sector 3 |
+----------+ +----------+ +----------+ +----------+
| Sector 0 | | Sector 1 | | Sector 0 | | Sector 1 |
+----------+ +----------+ +----------+ +----------+
   Disk 0       Disk 1       Disk 2       Disk 3
```

- 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

Raid level 2

```
+-------+ +-------+ +-------+ +-------+
|Byte3.0| |Byte3.1| |Byte3.0| |Byte3.1|
+-------+ +-------+ +-------+ +-------+
|Byte2.0| |Byte2.1| |Byte2.0| |Byte2.1|
+-------+ +-------+ +-------+ +-------+
|Byte1.0| |Byte1.1| |Byte1.0| |Byte1.1|
+-------+ +-------+ +-------+ +-------+
|Byte0.0| |Byte0.1| |Byte0.0| |Byte0.1|
+-------+ +-------+ +-------+ +-------+
 Disk 0    Disk 1    Disk 2    Disk 3
```

- 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 throughtput
- poco usata in pratica

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

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)

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

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)