Sistemi Operativi III

Giorgio Richelli

e-mail: giorgio_richelli@it.ibm.com

I file system Ext2 ed Ext3

Obiettivi del file system **Ext2** :

- Completa conformità alle “tradizionali” caratteristiche di file e file system UNIX;
- eccellenti prestazioni;
- stabilità;
- estendibilità.

Caratteristiche standard

- Supporto per
  - file regolari;
  - directory;
  - file speciali (*i.e.*, device);
  - link simbolici.
- Partizioni di grande formato
  (ordine di grandezza Terabyte, la dimensione esatta dipende da quella del blocco, ad esempio con blocchi da 1 KB la dimensione del filesystem può raggiungere i 4 TB).
- Nomi di file fino a 255 caratteri.
  - Estendibili a 1012 caratteri (modificando una #define).

Caratteristiche avanzate

- Gli attributi associati ai file:
  - permettono agli utenti di modificare il comportamento del kernel quando agisce su alcuni file;
  - possono essere associati a singoli file oppure a intere directory;
  - può essere selezione a tempo di mount la modalità di creazione dei file:
    - con la modalità BSD i file sono creati con lo stesso group id della directory che li contiene;
    - con la modalità System V i file sono creati con il group id della directory solo se questa ha il **setgid bit** attivo. Altrimenti i file sono creati con il group id del processo creante.

- può essere attivato un meccanismo di aggiornamento sincrono (*ala* BSD) per i "metadati" (inode, bitmap, ...) che però è normalmente disabilitato a cause delle scarse prestazioni.
- Ext2 permette di scegliere il blocco logico (tipicamente da 1 a 4 KB). Il miglioramento di prestazioni che in genere è reso possibile dalla scelta di un blocco più grande è comunque possibile attraverso le tecniche di preallocazione utilizzate dall'Ext2.
- Ext2 implementa i link simbolici *veloci* che mantengono il nome del file "reale" direttamente nell'inode.
  - il nome è però limitato a 60 caratteri in questo caso.

- Ext2 mantiene traccia dello stato del filesystem.
  - Uno speciale campo del superblocco è utilizzato dal kernel per indicare lo stato del filesystem:
  - un filesystem montato in modalità *read-write* è marcato come *not clean*;
  - quando è smontato o montato in *read-only* viene marcato come *clean*;
  - in caso di errori viene marcato come *erroneous*.
- Ext2 mantiene anche un contatore nel superblocco.
  Ogni volta che il file system è montato in modalità read-write, il contatore viene incrementato. Quando il contatore raggiunge un valore predefinito (registrato sempre nel superblocco) viene forzato un *check* del filesystem anche se questo appare nello stato *clean*.

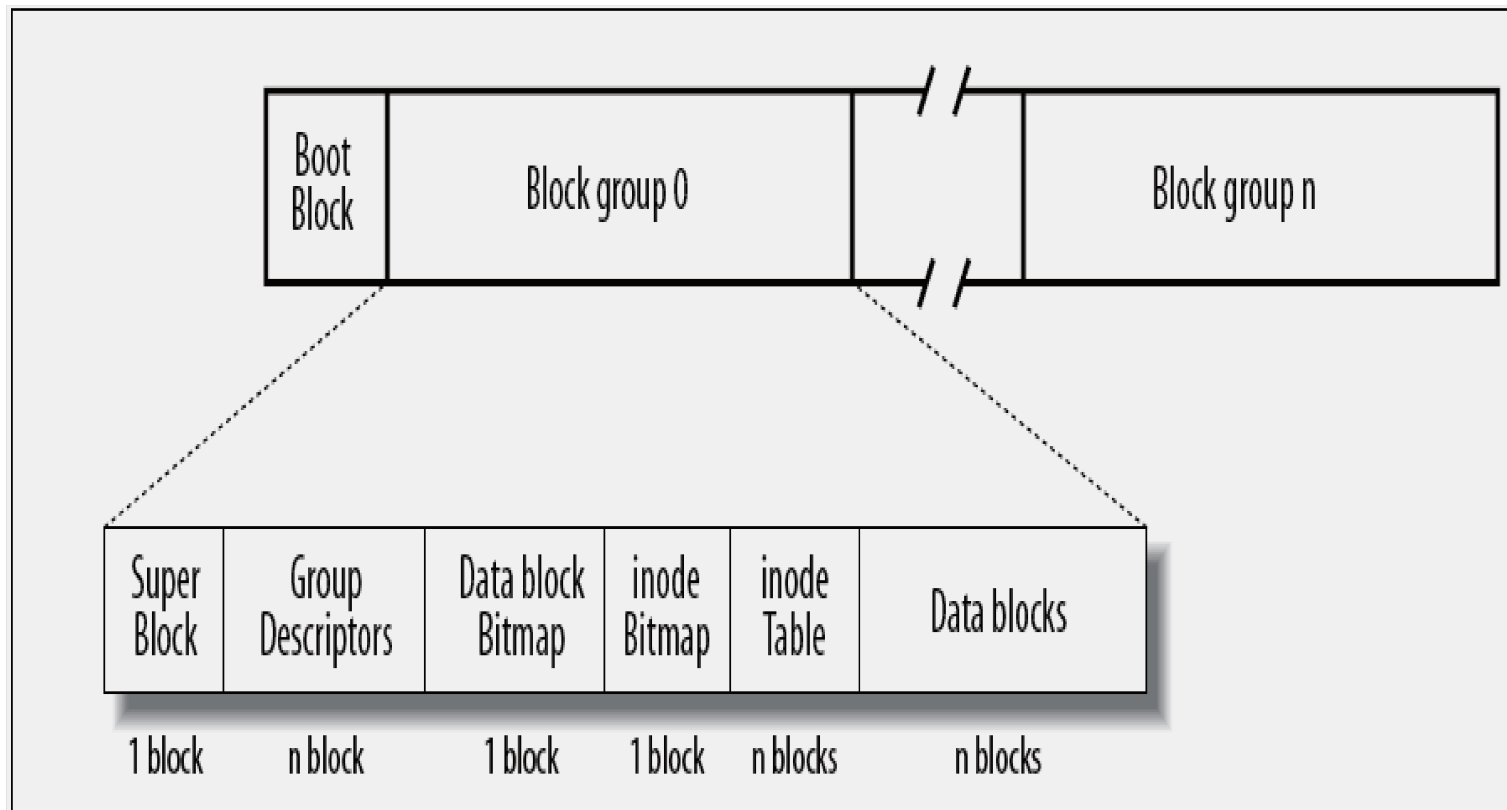
- Esistono infine due contatori nel superblocco che registrano il tempo dell'ultimo check ed il massimo intervallo di tempo tra due check.
  Quando il massimo intervallo è stato raggiunto, viene forzato un check indipendentemente dallo stato del filesystem.
- Il programma **tune2fs** permette di modificare:
  - il comportamento in caso di errore (girare il file system check, *panic*, ...);
  - il numero di operazioni di mount dopo il quale viene forzato il check;
  - l'intervallo di tempo massimo tra due check;
  - il numero di blocchi logici riservati per il super user (o un altro utente). Per *default* il 5% dei blocchi è riservato.

- In versioni precedenti, era possibile richiedere la cancellazione “sicura” di file.
  - sovrascrivendo con dati random i blocchi di disco del file che viene cancellato;
  - rende molto più difficoltoso il tentativo di recuperare i contenuti del file attraverso un *disk editor* software.
- Infine, Ext2 supporta alcune estensioni originariamente proposte all’interno del filesystem BSD 4.4:
  - file *immutabili*;
  - file escusivamente “estendibili”.

- Altre funzionalità potrebbero essere implementate in future versioni:
  - Files Compressi/Encrypted
  - Cancellazione *logica* dei files
    - permetterebbe di recuperare file eliminati
  - Frammentazione dei blocchi
    - file memorizzati in frammenti distinti dello stesso blocco
    - *unlikely*

Struttura fisica

La struttura fisica del filesystem Ext2 è fortemente influenzata da quella del filesystem BSD.



Il filesytem è costituito da *gruppi di blocchi* simili ai “gruppi di cilindri” nel BSD FFS.

- **Ogni** gruppo contiene una copia delle informazioni di controllo fondamentali del filesystem
  - superblocco;
  - descrittori del filesystem.

  ed una parte del filesystem:
  - bitmap dei blocchi;
  - bitmap degli inode;
  - parte degli inode;
  - parte dei blocchi dati.

  I gruppi di blocchi offrono vantaggi sia dal punto di vista dell’affidabilità (le informazioni fondamentali sono replicate) sia dal punto di vista delle prestazioni (riducendo la distanza tra la tabella degli inode ed i corrispondenti blocchi di dati).

- Sia il superblocco che i descrittori dei gruppi sono replicati ma, normalmentei, viene utilizzata solo la copia inclusa nel *block group* #0
- Quando viene effettuato un *fsck*, le informazioni vengono ri-duplicate
- In caso di inconsistenza, è possibile richiedere ad *e2fsck* di utilizzare una delle copie alternative
- Il numero totale di *block group* è approssimativamente: `S/(8*B)`, dove:
  - **S** è la dimensione della partizione
  - **B** è la dimensione del blocco

  Poichè la *block bitmap* deve risiedere in un singolo blocco

Le strutture dati dell'Ext2 sono definite in

`/usr/src/linux/include/linux/ext2_fs.h`

Il superblocco è descritto dalla seguente struttura:

```
struct ext2_super_block {
        __u32   s_inodes_count;         /* Inodes count */
        __u32   s_blocks_count;         /* Blocks count */
        __u32   s_r_blocks_count;       /* Reserved blocks count */
        __u32   s_free_blocks_count;    /* Free blocks count */
        __u32   s_free_inodes_count;    /* Free inodes count */
        __u32   s_first_data_block;     /* First Data Block */
        __u32   s_log_block_size;       /* Block size */
        __s32   s_log_frag_size;        /* Fragment size */
        __u32   s_blocks_per_group;     /* # Blocks per group */
        __u32   s_frags_per_group;      /* # Fragments per group */
        __u32   s_inodes_per_group;     /* # Inodes per group */
        __u32   s_mtime;                /* Mount time */
        __u32   s_wtime;                /* Write time */
        __u16   s_mnt_count;            /* Mount count */
        __s16   s_max_mnt_count;        /* Maximal mount count */
        __u16   s_magic;                /* Magic signature */
        __u16   s_state;                /* File system state */
        __u16   s_errors;               /* Behaviour when detecting errors */
```

```
        __u16   s_minor_rev_level;       /* minor revision level */
        __u32   s_lastcheck;             /* time of last check */
        __u32   s_checkinterval;         /* max. time between checks */
        __u32   s_creator_os;            /* OS */
        __u32   s_rev_level;             /* Revision level */
        __u16   s_def_resuid;            /* Default uid for reserved blocks */
        __u16   s_def_resgid;            /* Default gid for reserved blocks */
        __u32   s_first_ino;             /* First non-reserved inode */
        __u16   s_inode_size;            /* size of inode structure */
        __u16   s_block_group_nr;        /* block group # of this superblock */
        __u32   s_feature_compat;        /* compatible feature set */
        __u32   s_feature_incompat;      /* incompatible feature set */
        __u32   s_feature_ro_compat;     /* readonly-compatible feature set */
        __u8    s_uuid[16];              /* 128-bit uuid for volume */
        char    s_volume_name[16];       /* volume name */
        char    s_last_mounted[64];      /* directory where last mounted */
        __u32   s_algorithm_usage_bitmap; /* For compression */
        __u8    s_prealloc_blocks;       /* Nr of blocks to try to preallocate*/
        __u8    s_prealloc_dir_blocks;   /* Nr to preallocate for dirs */
        __u16   s_padding1;
        __u32   s_reserved[204];         /* Padding to the end of the block */
};
```

La seguente struttura descrive un *block descriptor*

```
struct ext2_group_desc
{
        __u32   bg_block_bitmap;        /* Blocks bitmap block */
        __u32   bg_inode_bitmap;        /* Inodes bitmap block */
        __u32   bg_inode_table;         /* Inodes table block */
        __u16   bg_free_blocks_count;   /* Free blocks count */
        __u16   bg_free_inodes_count;   /* Free inodes count */
        __u16   bg_used_dirs_count;     /* Directories count */
        __u16   bg_pad;
        __u32   bg_reserved[3];
};
```

Le directory nell'Ext2 sono implementate come liste collegate con elementi di lunghezza variabile:

```
|----------------------------------------------------------------|
|numero i-node|lunghezza entry|lunghezza nome|tipo file|nome file|
|----------------------------------------------------------------|
```

ad esempio:

```
|----------------|
|i1|16|05|1|file1|
|-------------------------|
|i2|24|14|1|long_file_name|
|-------------------------|
|i3|12|02|1|f2 |
|--------------|
```

In particolare i campi nella `ext2_dir_entry_2` sono i seguenti:

```
#define EXT2_NAME_LEN 255

struct ext2_dir_entry {
        __u32   inode;                  /* Inode number */
        __u16   rec_len;                /* Directory entry length */
        __u8    name_len;               /* Name length */
        __u8    file_type;
        char    name[EXT2_NAME_LEN];    /* File name */
};
```

La lunghezza di una entry è sempre un multiplo di 4.
I possibili tipi di file sono:

```
enum {
        EXT2_FT_UNKNOWN, EXT2_FT_REG_FILE, EXT2_FT_DIR,  EXT2_FT_CHRDEV,
        EXT2_FT_BLKDEV,  EXT2_FT_FIFO,     EXT2_FT_SOCK, EXT2_FT_SYMLINK,
        EXT2_FT_MAX
};
```

Esempio:

```
$ ls -1a /home/giorgio
.
..
.bash_profile
.bashrc
mbox
public_html
tmp
```

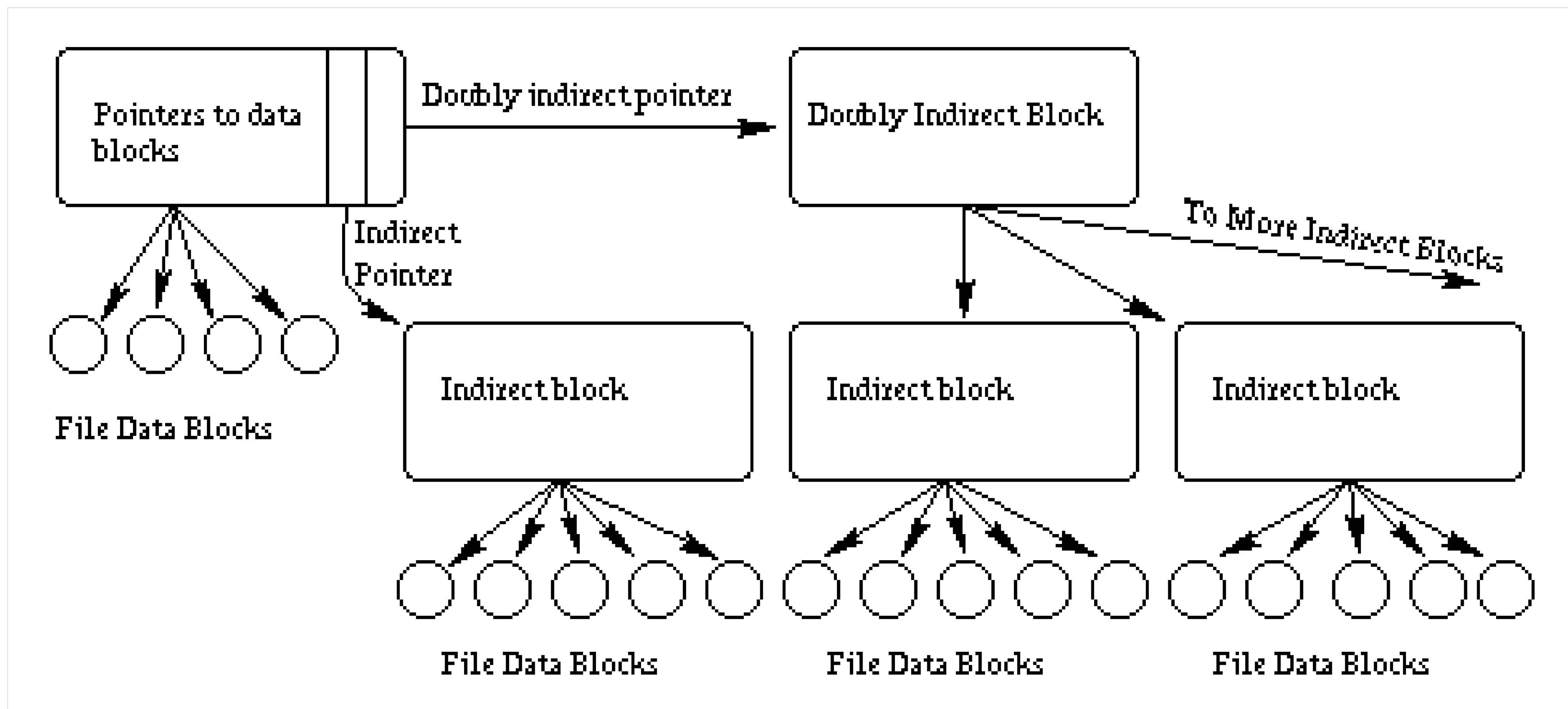
```
offset  size    description
------- ------- -----------
      0       4 inode number (783362)
      4       2 record length (9)
      6       1 name length (1)
      7       1 file type (EXT2_FT_DIR)
      8       1 name (.)

      9       4 inode number (1109761)
     13       2 record length (10)
     15       1 name length (2)
     16       1 file type (EXT2_FT_DIR)
     17       2 name (..)

     19       4 inode number (783364)
     23       2 record length (21)
     25       1 name length (13)
     26       1 file type (EXT2_FT_REG_FILE)
     27      13 name (.bash_profile)

     40       4 inode number (783363)
     44       2 record length (15)
     46       1 name length (7)
```

```
47        1 file type (EXT2_FT_REG_FILE)
48        7 name (.bashrc)

55        4 inode number (783377)
59        2 record length (12)
61        1 name length (4)
62        1 file type (EXT2_FT_REG_FILE)
63        4 name (mbox)

67        4 inode number (783545)
71        2 record length (19)
73        1 name length (11)
74        1 file type (EXT2_FT_DIR)
75       11 name (public_html)

86        4 inode number (669354)
90        2 record length (11)
92        1 name length (3)
93        1 file type (EXT2_FT_DIR)
94        3 name (tmp)

97        4 inode number (0)
101       2 record length (3999)
```

```
103       1 name length (0)
104       1 file type (EXT2_FT_UNKNOWN)
105       0 name ()
```

```
struct ext2_inode {
        __u16   i_mode;         /* File mode */
        __u16   i_uid;          /* Low 16 bits of Owner Uid */
        __u32   i_size;         /* Size in bytes */
        __u32   i_atime;        /* Access time */
        __u32   i_ctime;        /* Creation time */
        __u32   i_mtime;        /* Modification time */
        __u32   i_dtime;        /* Deletion Time */
        __u16   i_gid;          /* Low 16 bits of Group Id */
        __u16   i_links_count;  /* Links count */
        __u32   i_blocks;       /* Blocks count */
        __u32   i_flags;        /* File flags */
        union   osd1            /* Special operating system information */
        __u32   i_block[EXT2_N_BLOCKS];/* Pointers to blocks, EXT2_N_BLOCKS=15*/
        __u32   i_generation;   /* File version (for NFS) */
        __u32   i_file_acl;     /* File ACL */
        __u32   i_dir_acl;      /* Directory ACL */
        __u32   i_faddr;        /* Fragment address */
        union   osd2            /* Special operating system information 2 */
}
```

Indirizzo dei blocchi dall'inode



- La dimensione del blocco influenza la massima dimensione possibile per un file

Block Size	Direct	1 Indirect	2-Indirect	3-Indirect
1024	12 KB	268 KB	63.55 MB	2 GB
2048	24 KB	1.02 MB	512.02 MB	2 GB
4096	48 KB	4.04 MB	2 GB	-

- Il campo **`i_flags`** della struttura inode, permette di attivare alcune funzionalità del filesystem

Name	Value	Description
EXT2_SECRM_FL	0x00000001	secure deletion
EXT2_UNRM_FL	0x00000002	record for undelete
EXT2_COMPR_FL	0x00000004	compressed file
EXT2_SYNC_FL	0x00000008	synchronous updates
EXT2_IMMUTABLE_FL	0x00000010	immutable file
EXT2_APPEND_FL	0x00000020	append only
EXT2_NODUMP_FL	0x00000040	do not dump/delete file
EXT2_NOATIME_FL	0x00000080	do not update .i_atime
EXT2_DIRTY_FL	0x00000100	dirty (file is in use?)
EXT2_COMPRBLK_FL	0x00000200	compressed blocks
EXT2_NOCOMPR_FL	0x00000400	access raw compressed data
EXT2_ECOMPR_FL	0x00000800	compression error
EXT2_BTREE_FL	0x00010000	b-tree format directory
EXT2_INDEX_FL	0x00010000	Hash indexed directory
EXT2_IMAGIC_FL	0x00020000	?
EXT3_JOURNAL_DATA_FL	0x00040000	journal file data
EXT2_RESERVED_FL	0x80000000	reserved for ext2 implementation

Improved Directory Handling

- La normale gestione (a lista) delle directory può diventare molto lenta al crescere del numero di file
- Ext2 può implementare meccanismi più efficienti, ad esempio il formato *Indexed Directory*:
  - Compute a hash of the name
  - Read the index root
  - Use binary search (linear in the current code) to find the first index or leaf block that could contain the target hash (in tree order)
  - Repeat the above until the lowest tree level is reached
  - Read the leaf directory entry block and do a normal Ext2 directory block search in it.
  - If the name is found, return its directory entry and buffer
  - Otherwise, if the collision bit of the next directory entry is set, continue searching in the successor block

Tempo richiesto per la creazione di `N` files in una directory:

N	Indexed	Normal
10000	0m1.350s	0m23.670s
20000	0m2.720s	1m20.470s
30000	0m4.330s	3m9.320s
40000	0m5.890s	5m48.750s
50000	0m7.040s	9m31.270s
60000	0m8.610s	13m52.250s
70000	0m9.980s	19m24.070s
80000	0m12.060s	25m36.730s
90000	0m13.400s	33m18.550s

Ottimizzazione delle prestazioni

- Ext2 utilizza un meccanismo di *read* “in avanti”: quando deve essere letto un blocco, vengono in effetti letti più blocchi consecutivi in modo da aumentare la probabilità che una successiva operazione di lettura trovi il blocco già disponibile. Questo meccanismo è utilizzato sia per la lettura di file che di directory.
- Per quanto riguarda l’allocazione, Ext2 cerca, per quanto possibile, di allocare i blocchi dati di un file nello stesso gruppo dell’inode.
- Il “successore” di un blocco è allocato cercando il primo blocco libero entro i successivi 32 blocchi. Se la ricerca fallisce, cerca in avanti:

1. un byte libero nella bitmap (8 blocchi liberi);
2. un qualsiasi bit libero nella bitmap del gruppo;
3. nei successivi gruppi seguendo la stessa logica.

- Quando scrive dati in un file, Ext2 prealloca fino ad 8 blocchi adiacenti. In questo modo vengono velocizzate anche le successive operazioni di lettura sequenziale del file.
  – i blocchi preallocati sono liberati se il file viene chiuso oppure se la successiva allocazione non è sequenziale.
- La consistenza delle directory è protetta dando ai corrispondenti metadati una priorità più alta nella scrittura asincrona.

La libreria Ext2fs

Per facilitare l'accesso e la manipolazione delle strutture di controllo dell'Ext2, è stata sviluppata la libreria **libext2fs** disponibile insieme ad un set di strumenti che la utilizzano da `http://e2fsprogs.sourceforge.net/`.

- La libreria è disegnata per facilitare il riutilizzo di codice fornendo astrazioni software che permettono, ad esempio, di invocare una funzione specificata dall'utente su tutti i blocchi di disco associati ad un inode (funzione `ext2fs_block_iterate`) oppure su tutti i file di una directory.
- La libreria permette diversi tipi di operazioni:
  1. orientate al filesystem:
     - apertura/chiusura del filesystem;

- lettura/scrittura delle bitmap;
- creazione di un nuovo filesystem su disco
- manipolazione della lista dei *bad block*.

2. operazioni sulle directory:
   - creazione ed "espansione" di directory;
   - creazione e rimozione di entry nella directory;
   - risoluzione path→inode ed inversa (inode→path).
3. operazioni sugli inode:
   - scansione della tabella degli inode;
   - lettura e scrittura degli inode;
   - allocazione e liberazione di blocchi e inode;
   - itera su ogni blocco di un inode.

Strumenti per l'Ext2

Sono disponibili un set di programmi per la gestione diretta di un file system Ext2:

- `mke2fs`: inizializza una partizione per contenere un filesystem Ext2;
- `tune2fs`: cambia il comportamento in caso di errore, il massimo numero di mount prima di un check, il numero di blocchi logici riservati per un utente (tipicamente il super user);
- `dumpe2fs`: permette di visualizzare la struttura di un filesystem e le informazioni contenute nel superblocco

- Il tool più interessante è ovviamente `e2fsck`, il *filesystem checker* che utilizza, ovviamente, i servizi della libreria ext2fs.
  `e2fsck` effettua una serie di “passi” sul filesystem:
  1. itera su tutti gli inode ed esegue controlli su ognuno come “oggetto indipendente”, senza cioè richiedere controlli incrociati su altri oggetti del filesystem.
     Ad esempio, controlla che la modalità di accesso al file sia legale oppure che tutti i blocchi di disco appartenenti all'inode siano numeri di blocco legali.
     Viene inoltre definita una bitmap di quali blocchi ed inode sono in uso.
     Se vengono rilevati blocchi di dati associati a più di un inode, si effettuano dei passi aggiuntivi (**1B-1D**) per risolvere questi conflitti o “clonando” i blocchi dati in modo che ogni inode abbia una propria copia, oppure disassociandoli da uno o più degli inode.

2. Vengono controllate “indipendentemente” le directory in modo da minimizzare il numero di accessi al disco. In particolare si controlla se le entry si riferiscono ad inode che risultano in uso sulla base dell’analisi effettuata al passo 1.
   Viene controllata l’esistenza delle entry `.` e `..` e che l’inode per l’entry `.` coincida con la directory corrente.
   Il passo 2 registra anche informazioni relative alla directory padre.
   Alla fine del passo 2 sono state effettuate praticamente tutte le operazioni di accesso a disco richieste.
   Gli altri passi utilizzano informazioni salvate in memoria durante i passi 1 e 2 e sono per lo più *CPU bound*.
3. Viene controllata la connettività delle directory.
   `e2fsck` traccia all’indietro il path di ogni directory fino alla root usando informazioni registrate al passo 2. Viene controllata a questo passo la validità della entry `..` di ogni directory. Le

directory che non possono essere fatte risalire fino alla root sono associate alla directory speciale **/lost+found**.

4. `e2fsck` controlla il reference count di tutti gli inode, iterando su tutti e paragonando il link count salvato al passo 1 a contatori interni calcolati durante i passi 2 e 3.
   Tutti i file non cancellati con un link count uguale a zero sono mossi nella directory **/lost+found**.
5. `e2fsck` controlla la validità delle informazioni globali del filesystem. In particolare confronta le bitmap di blocchi e inode costruite ai passi precedenti con quelle disponibili sul filesystem modificando le copie su disco, se necessario.

- Un altro utile strumento è `debugfs` che può essere utilizzato per esaminare e modificare lo stato del filesystem.
  – può essere visto come un'interfaccia interattiva alla libreria **ext2fs** che trasla i comandi inseriti dall'utente.

`debugfs` apre il filesystem in modalità *read-only* a meno che non venga esplicitamente utilizzata l'opzione **-w**.

- `chattr` e `lsattr` consentono di definire gli attributi dei file (+ attiva, `-` disattiva):
  - **A**: non aggiorna `atime`;
  - **a**: solo *append*;
  - **S**: aggiornamento sincrono;
  - **i**: file immutabile;
  - **j**: attivazione *journaling*;
  - **c, u, s**: non implementati.

Deframmentazione in Ext2

Strutturata in 4 fasi:

1. Scandisce gli inode per raccogliere informazioni.
2. Ordina gli inode in modo da ridurre la distanza di cui dovranno muoversi i blocchi durante la rilocazione.
3. Crea le liste di rilocazione (in avanti ed inversa) scandendo nuovamente gli inode.
4. Riordina le aree dati su disco sulla base delle mappe di rilocazione.

Affidabilitá e Predicibilitá

- Ext2 offre buone caratteristiche di affidabilitá ma non é *predicibile*
- La fase di recovery dovrebbe essere in grado di stabilire quali modifiche erano in corso al momento del crash
- La predicibilitá richiederebbe, in generale, che le operazioni di scrittura siano fatte in un ordine specifico ogni qualvolta venga modificato piú di un blocco di metadati.

Affidabilitá e Predicibilitá

- Esistono vari modi per ottenere l'ordinamento fra le write
- Il piú semplice é quello di garantire che un'operazione sia completata prima di iniziarne un'altra
  - BSD FFS *Synchronous Metadata Update*
  - Impedisce di raggruppare le transazioni
  - Forte impatto sulle prestazioni

Affidabilitá e Predicibilitá

- *Deferred Order Write*
  - Mantenere un'ordinamento sui buffer in memoria e scrivere un blocco solo dopo aver scritto tutti i suoi predecessori
  - Soggetto a deadlock (e.g. spostare un file fra A e B ed un'altro da B ad A)
  - FreeBSD implementa le *soft updates*
    - Rollback delle modifiche nel blocco dipendono da altre
    - Le modifiche vengono riapplicate dopo che le dipendenze sono state soddisfatte

Filesystem “Journaling” ed Ext3

Journaling significa salvare (su disco) la storia di tutti i cambiamenti del file system in una sorta di registro.

- Il filesystem deve effettuare le operazioni di scrittura su disco in un ordine predefinito ogni volta che un'operazione modifica più blocchi su disco.
- Il *journal* registra i nuovi contenuti dei blocchi di metadati mentre completa le transazioni.
- Vengono registrati tre tipi di dati:
  - metadati;
  - blocchi dei descrittori di gruppo;
  - blocchi di header del filesystem.

Ext3 è un file system **completamente** compatibile con Ext2 (sono derivati esattamente dallo stesso source) che offre supporto per il *journaling*.

- Il problema principale con Ext2 è che `fsck` può richiedere un tempo molto lungo su filesystem di grandi dimensioni (ore!).
- L'idea di Ext3 è di aggiungere esclusivamente il *journaling* in modo che, se questa funzionalità non viene utilizzata, si possa utilizzare semplicemente come un filesystem Ext2 "standard".
- Il modulo di *journaling* è in realtà indipendente dal filesystem EXT3.
  Può essere visto come un livello generico (**JFS**) che permette di effettuare modifiche nella *buffer cache* secondo una logica "transazionale".
  - Ext3 si limita ad implementare le operazioni come transazioni.

- Esempio:
  - Ext3 deve effettuare modifiche a 5 blocchi di disco (metadati).
  - Ext3 registra l'operazione come una transazione.
  - Il modulo di *journaling* garantisce che dopo un reboot sono stati modificati tutti i 5 blocchi di disco, oppure nessuno.
- Naturalmente, durante la transazione, è necessario mantenere due copie dei dati in corso di aggiornamento.
  - Diverse possibili tecniche di implementazione:
    - Redo
    - Undo
    - *Log File Systems*

- JBD é il modulo di *journaling* che fornisce un'API che implementa il concetto di transazione.
- Un nuovo *journal* può essere registrato con il relativo modulo specificando (nel superblocco) quale inode costituisce il *log*.
- Due diversi layer, implementati in `fs/jbd` e `fs/ext3`
- `ext3` utilizza il `jbd` per:
  - iniziare e terminare una transazione
  - richiedere un recovery
- `ext3` usa un *physical* journaling
  - Viene memorizzato sul journal l'intero blocco modificato
  - Nel *jogical* journaling sono registrate solo le modifiche
- A differenza di altre implementazioni, `ext3` non impone (in teoria) come i dati saranno memorizzati su disco

- JBD fornisce un API che puó essere usata per implementare il journaling. Finora, solo ext3, ext4 (JBD2), e OCFS2 (Oracle Cluster File System) usano JBD.
- JBD raggruppa diverse operazioni *atomiche* in una singola transazione, che verrá scritta nel journal dopo un certo intervallo oppure se il journal risulta pieno.
- Una transazione puó trovarsi in uno dei seguenti stati:

Running La transazione é attiva ed accetta ulteriori operazioni.

Locked Non accetta piú transazioni, ma quelle esistenti non sono ancora complete.

Flush La transazione é completa e viene scritta nel journal

Commit Le modifiche sono applicate al filesystem.

Finished La transazione é stata completamente scritta nel filesystem. Puó essere cancellata dal journal.

- Tre diversi tipi di journaling possibili:

  Writeback Vengono memorizzati nel journal solo i metadati.
  Non esiste alcun ordinamento fra i dati ed i metadati
  È Fornisce le migliori prestazioni, ma pu produrre perdita di dati

  Ordered Come il precedente, ma vengono scritti i blocchi modificati prima di inserire il *commit* della transazione
  È il default. Minimizza (ma non elimina) la possibilità di perdere dati

  Journal Sia i dati che i metadati sono memorizzati nel journal
  In generale (ma non sempre) ha il maggiore overhead.

Le “fasi” del journaling

Se devono essere aggiornati i metadati su disco:

1. i metadati sono registrati in un’apposita area riservata per il *journal*;
2. alla fine di questa *transazione* viene aggiunto un “record di completamento” nel *journal*;
3. dopo la scrittura **su disco** del record di completamento, i nuovi metadati sono copiati nella locazione standard.

In caso di crash, esistono due possibilità, a seconda che il record di *commit* appaia o meno nel log:

- Nel primo caso, la transazione è completa ed i nuovi metadati possono essere comunque copiati dalla locazione riservata, utilizzata nella fase 1, per il *journal*.
- Nel secondo, invece, i nuovi metadati possono essere semplicemente ignorati.

Questa operazione è molto più veloce del classico `fsck` perché il *journal* è piccolo paragonato all'intero filesystem.

- L'API di journaling:
  - definisce funzioni di *start()*, *stop()* di una transazione
  - permette il *nesting* di transazioni
  - consente di riservare spazio nel log (deve essere sufficiente per memorizzare la transazione)
  - non è previsto l'*abort* di una transazione

“Anatomia” di una transazione

Qualsiasi singola richiesta fatta da un’applicazione verso il filesystem risulta in una transazione.

Ad esempio una scrittura su file implica l’aggiornamento:

- del relativo tempo nell’inode del file su disco;

ma anche (molto probabilmente):

- della quantità di spazio libero disponibile;
- delle quota;
- della bitmap dei blocchi liberi.

È quindi necessario definire relazioni tra le transazioni.

Immaginiamo una situazione in cui:

1. Una transazione **A** modifica un blocco di disco.
2. Una transazione **B** legge il nuovo blocco ed aggiorna il disco (non necessariamente lo stesso blocco) sulla base di quanto letto.

È chiaro che la transazione **A** non dovrebbe essere registrata **dopo** la transazione **B**.

Un *journaled filesystem* sfrutta concetti e tecnologie in gran parte derivate dal mondo dei database ma con alcune importanti differenze.

- non è necessario supportare l'annullamento di una transazione;
  - quando la transazione inizia, tutti i controlli necessari sono già stati effettuati.
- le transazioni hanno una durata (generalmente) breve:
  - è possibile registrare le transazioni nell'ordine sequenziale in

cui avvengono senza particolare impatto (negativo) sulle prestazioni.

Queste osservazioni permettono di semplificare notevolmente il meccanismo di gestione delle transazioni:

- invece di creare una transazione separata per ogni aggiornamento, più operazioni sono associate ad una singola transazione *system-wide*.
- come spesso accade, è necessario individuare il giusto compromesso tra le prestazioni
  (per cui la transazione dovrebbe estendersi per tempi lunghi) e l'affidabilità
  (per cui la transazione dovrebbe essere breve).

Formato del *journal*

Nel *journal* vengono registrati 3 tipi di dati:

1. metadati dei file;
2. blocchi di *descrittori*;
3. blocchi di *header*.

Un blocco di metadati contiene le informazioni sui metadati coinvolti in una transazione.

- anche un piccolo cambio ai metadati, implica la scrittura di un blocco del *journal*;
- l'impatto sulle prestazioni è comunque limitato dato che la maggior parte delle scritture sono sequenziali e possono essere quindi gestite efficentemente dal controller del disco.

I blocchi di descrittori sono blocchi del *journal* che descrivono blocchi di metadati.

- In particolare contengono le informazioni sui blocchi di disco del filesystem su cui vanno aggiornati i metadati.

Sia i blocchi dei metadati che i descrittori sono scritti sequenzialmente in modo circolare
(quando si raggiunge la fine del journal, si inizia da capo).

- Si mantiene un puntatore all'ultimo blocco scritto (testa) ed uno all'ultimo blocco che **non** è stato sovrascritto (coda).

I blocchi *header* si trovano in locazioni fissate e mantengono i puntatori alla “testa” ed alla “coda” più un numero di sequenza.

- in caso di problemi, vengono scanditi i blocchi *header* e selezionato quello con il numero di sequenza più alto;
- i relativi puntatori alla testa ed alla coda sono utilizzati per il recupero delle transazioni, partendo dalla coda fino alla testa.

Sincronizzazione del *journal*

1. la transazione corrente viene chiusa;
   le operazioni incomplete rimangono associate alla transazione chiusa;
2. inizia la trascrizione sul *journal* dei metadati modificati durante la transazione;
   contemporaneamente vengono scritti anche i blocchi dati corrispondenti alla transazione;
3. attende il completamento delle operazioni sul filesystem che appartengono alla transazione;
4. attende la registrazione di tutti gli aggiornamenti appartenenti alla transazione sul *journal*;
   - il *commit* è costituito da un singolo blocco da 512 bytes

5. aggiorna il blocco *header* per registrare la nuova testa e coda del log salvando la transazione su disco;

6. i blocchi del *journal* utilizzati per la transazione possono essere riutilizzati solo quando sono stati **tutti** copiati su disco.
   A questo punto viene registrata la nuova posizione della "coda" del *journal*.

Per migliorare le prestazioni è consentito che ci sia "concorrenza" tra due consecutive transazioni:

- mentre si registra una transazione, gli aggiornamenti del filesystem vengono associati ad una nuova transazione;

- per evitare possibili collisioni nel caso in cui due aggiornamenti vanno a modificare lo stesso buffer, viene creata una copia del buffer "conteso".

Superblocco EXT3

- La struttura del superblocco EXT3 è identica a quella EXT2, per la prima parte, a cui segue:

```
        ...
        __u16   s_padding1;
        /*
         * Journaling support valid if EXT3_FEATURE_COMPAT_HAS_JOURNAL set.
         */
/*D0*/  __u8    s_journal_uuid[16];      /* uuid of journal superblock */
/*E0*/  __u32   s_journal_inum;          /* inode number of journal file */
        __u32   s_journal_dev;           /* device number of journal file */
        __u32   s_last_orphan;           /* start of list of inodes to delete */

/*EC*/  __u32   s_reserved[197];         /* Padding to the end of the block */
};
```

- Alcuni flag impediscono il mount come EXT2, nel caso in cui sia necessario un recovery del journal