

EXT2 FileSystem

馬紀哲

macc@csie.nctu.edu.tw



What is EXT2 ?

- | Ext2fs was designed by Remy Card as an extensible and powerful filesystem for Linux
- | This new filesystem removed the two big Minix limitations: its maximal size was 2 giga bytes and the maximal file name size was 255 characters

General Characteristic

- | Block size : from 1024 to 4096 bytes
- | Inodes (index-nodes) :
 - >depending on the expected # of files
 - >df -i, df -a

EX.

DISK 4096000KB , Inodes 1000

General Characteristic

- | Preallocation : reduce fragmentation
- | Block groups : adjacent blocks
- | for data blocks → the file's inode
for inode (nondirectory) → file's parent dir.
(directory) → not kept together with parent dir.,
but rather dispersed throughout the available block groups
- | Fast symbolic links : pathname < 60B stored in
inode can be translated without reading block

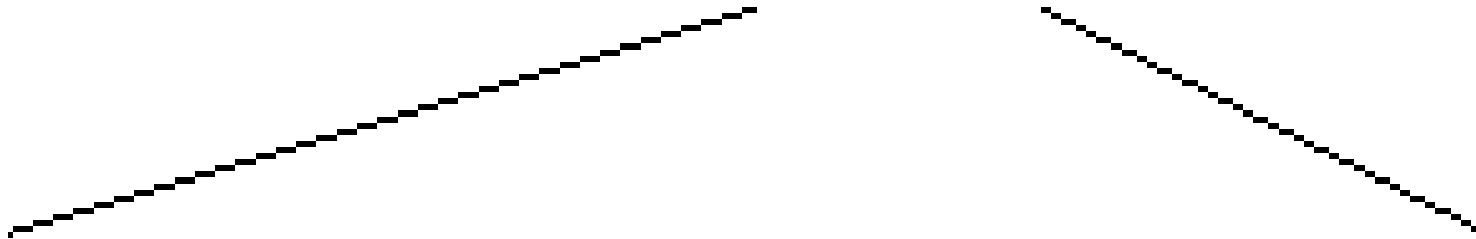
Other features

Make it both robust and flexible

- | File-updating strategy
- | Automatic consistency check (/sbin/e2fsck)
- | Immutable files (even the superuser can't modify it)

Disk Data Structures

I EXT2 partition



Super Block	Group Description	Block Bitmap	Inode Bitmap	Inode Table	Data Blocks
-------------	-------------------	--------------	--------------	-------------	-------------

I How many block groups are there ?

each block group there can be at most $8xb$ blocks, where b is the block size in bytes. The total # of block groups is roughly $s/(8xb)$, s is the partition size in block.

EX. 8GB partition with a 4KB block size.
64 block groups!

Superblock

```
| 339 struct ext2_super_block {
| 340     __u32     s_inodes_count;           /* Inodes count */
| 341     __u32     s_blocks_count;         /* Blocks count */
| 342     __u32     s_r_blocks_count;       /* Reserved blocks count */
| 343     __u32     s_free_blocks_count;    /* Free blocks count */
| 344     __u32     s_free_inodes_count;    /* Free inodes count */
| 345     __u32     s_first_data_block;     /* First Data Block */
| 346     __u32     s_log_block_size;      /* Block size */
| 347     __s32     s_log_frag_size;       /* Fragment size */
| 348     __u32     s_blocks_per_group;     /* # Blocks per group */
| 349     __u32     s_frags_per_group;      /* # Fragments per group */
| 350     __u32     s_inodes_per_group;     /* # Inodes per group */
| 351     __u32     s_mtime;                /* Mount time */
| 352     __u32     s_wtime;                /* Write time */
| 353     __u16     s_mnt_count;            /* Mount count */
| 354     __s16     s_max_mnt_count;        /* Maximal mount count */
| 355     __u16     s_magic;                /* Magic signature */
| 356     __u16     s_state;                /* File system state */
| 357     __u16     s_errors;               /* Behaviour when detecting errors */
| 358     __u16     s_minor_rev_level;      /* minor revision level */
| 359     __u32     s_lastcheck;           /* time of last check */
| 360     __u32     s_checkinterval;       /* max. time between checks */
```

Superblock

```
| 361  __u32    s_creator_os;          /* OS */
| 362  __u32    s_rev_level;          /* Revision level */
| 363  __u16    s_def_resuid;         /* Default uid for reserved blocks */
| 364  __u16    s_def_resgid;         /* Default gid for reserved blocks */
| 378  __u32    s_first_ino;         /* First non-reserved inode */
| 379  __u16    s_inode_size;        /* size of inode structure */
| 380  __u16    s_block_group_nr;    /* block group # of this superblock */
| 381  __u32    s_feature_compat;    /* compatible feature set */
| 382  __u32    s_feature_incompat;  /* incompatible feature set */
| 383  __u32    s_feature_ro_compat; /* readonly-compatible feature set */
| 384  __u8     s_uuid[16];          /* 128-bit uuid for volume */
| 385  char     s_volume_name[16];   /* volume name */
| 386  char     s_last_mounted[64];  /* directory where last mounted */
| 387  __u32    s_algorithm_usage_bitmap; /* For compression */
| 392  __u8     s_prealloc_blocks;    /* Nr of blocks to try to preallocate*/
| 393  __u8     s_prealloc_dir_blocks; /* Nr to preallocate for dirs */
| 394  __u16    s_padding1;
| 395  __u32    s_reserved[204];     /* Padding to the end of the block */
| 396  };
```

Group Descriptor and Bitmap

```
| 148 struct ext2_group_desc
| 149 {
| 150  __u32    bg_block_bitmap;          /* Blocks bitmap block */
| 151  __u32    bg_inode_bitmap;        /* Inodes bitmap block */
| 152  __u32    bg_inode_table;         /* Inodes table block */
| 153  __u16    bg_free_blocks_count; /* Free blocks count */
| 154  __u16    bg_free_inodes_count; /* Free inodes count */
| 155  __u16    bg_used_dirs_count;    /* Directories count */
| 156  __u16    bg_pad;
| 157  __u32    bg_reserved[3];
| 158  };
```

Inode Table

```
217 struct ext2_inode {
218     __u16    i_mode;           /* File mode */
219     __u16    i_uid;           /* Low 16 bits of Owner Uid */
220     __u32    i_size;          /* Size in bytes */
221     __u32    i_atime;        /* Access time */
222     __u32    i_ctime;        /* Creation time */
223     __u32    i_mtime;        /* Modification time */
224     __u32    i_dtime;        /* Deletion Time */
225     __u16    i_gid;           /* Low 16 bits of Group Id */
226     __u16    i_links_count;   /* Links count */
227     __u32    i_blocks;       /* Blocks count */ /* I_size and I_block */
228     __u32    i_flags;        /* File flags */
240     __u32    i_block[EXT2_N_BLOCKS]; /* Pointers to blocks */
241     __u32    i_generation; /* File version (for NFS) */
242     __u32    i_file_acl;     /* File ACL */
243     __u32    i_dir_acl;     /* Directory ACL */
244     __u32    i_faddr;        /* Fragment address */
268     } osd2;                  /* OS dependent 2 */
269 };
```

File Holes

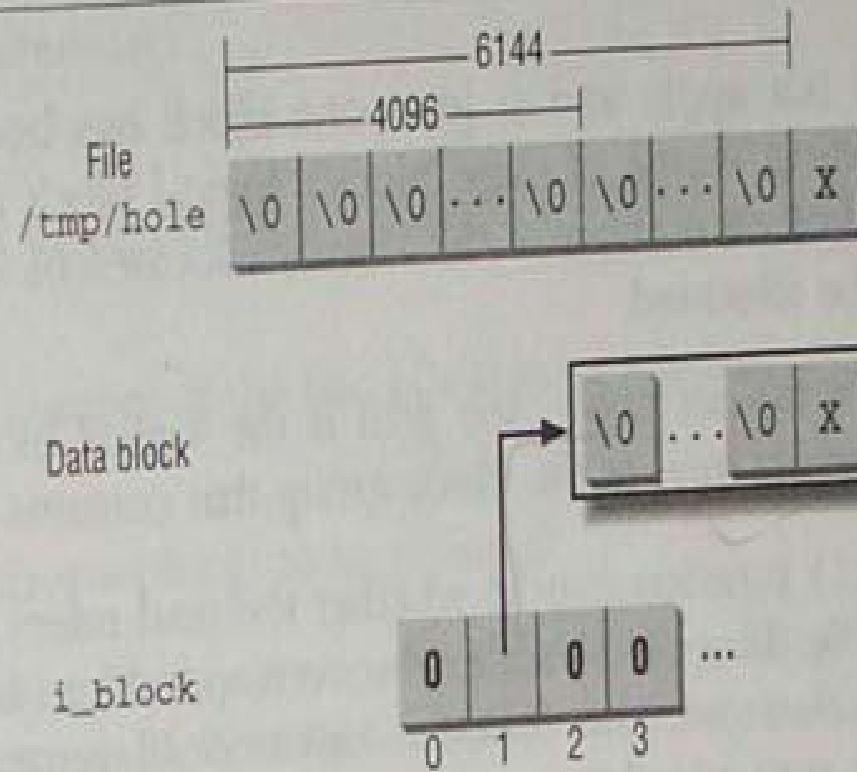


Figure 17-6. A file with an initial hole



I How Various File Types Use Disk Blocks

1. Regular file
2. Directory
3. Symbolic link
4. Device file, pipe, and socket

Directory

	inode	rec_len	file_type	name_len	name
0	21	12	1	2	. \0 \0 \0
12	22	12	2	2	. . \0 \0
24	53	16	5	2	h o m e 1 \0 \0 \0
40	67	28	3	2	u s r \0
52	0	16	7	1	o l d f i l e \0
68	34	12	4	2	s b i n

Figure 17-2. An example of EXT2 directory

Memory Data Structures

- | Created new file
(s_free_inodes_count in superblock)
(bg_free_inodes_count in block des.)
- | Appends some data to an existing file
(s_free_blocks_count in superblock)
(bg_free_blocks_count in block des.)
- | Rewriting a existing file
(s_wtime in superblock)



I The ext2_sb_info

VFS superblock objects (u field) :

1. Most of the disk superblock fields

2. Bitmap caches

3. s_sbh points

4. s_es points

5. s_desc_per_block

6. s_group_desc

Memory data structure

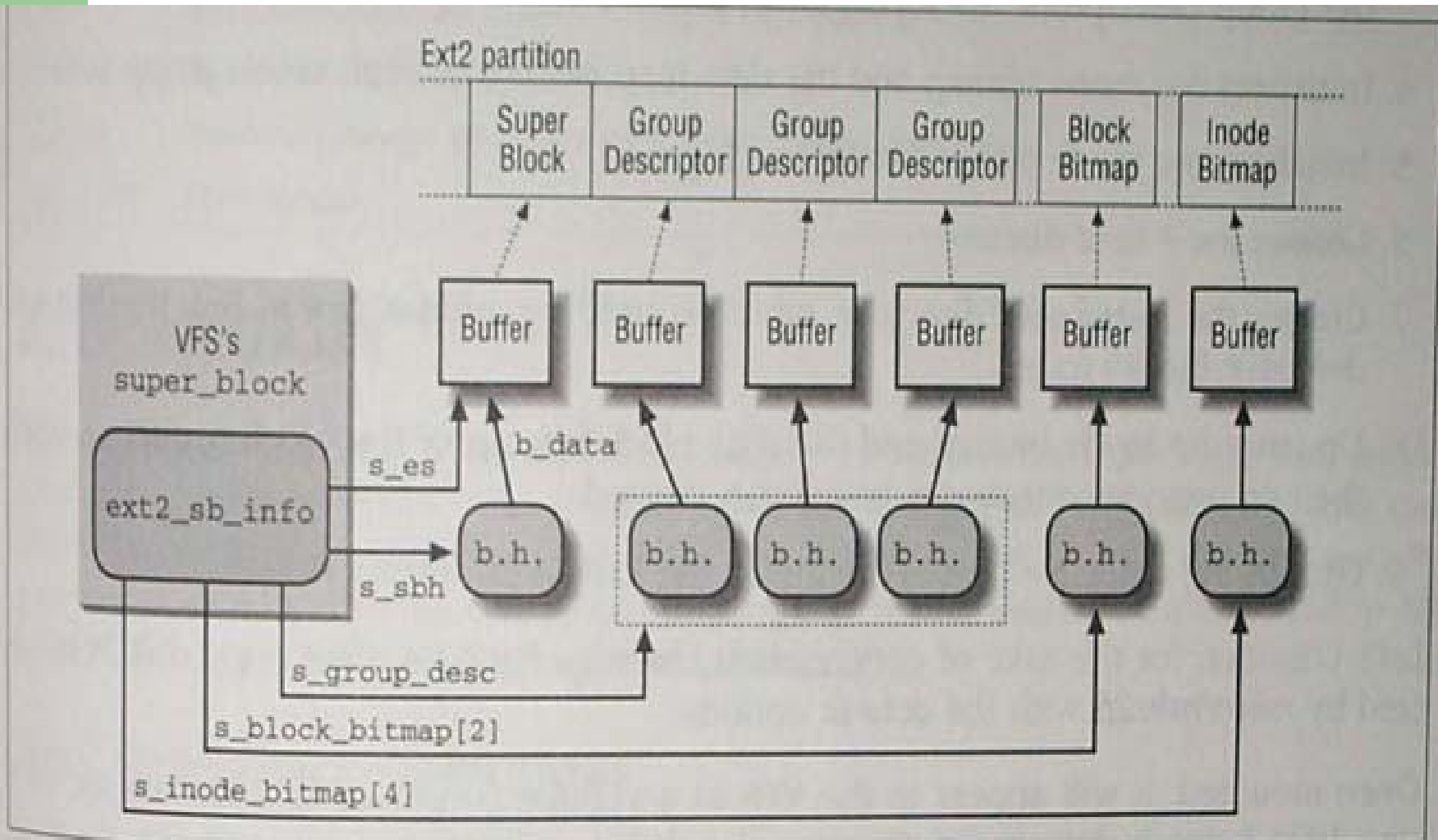


Figure 17-4. Ext2 memory data structures



I Bitmap Caches

I What's the problem ?

4GB disk, 1KB block size

require to store all 1024 bitmaps in memory

Managing Disk Space

- | Creating Inodes
- | Deleting Inodes
- | Data Blocks Addressing
- | Allocating a Data Block
- | Releasing a Data Block

I Creating Inodes (ext2_new_inode())

get_empty_inode()	lock_super()
is a directory ?	load_inode_bitmap
ll_rw_blocks()	dec. bg_free_inodes_count
dec. s_free_inodes_count	initialize the fields of the inode object
insert inode object into inode_hashtable	mark_inode_dirty
unlock_superblock	returns the addr. of the new inode

Creating Inodes

ext2_new_inode()

```
l 280 struct inode * ext2_new_inode (const struct inode * dir, int mode, int * err)
l 281 {
l 282     struct super_block * sb;
l 283     struct buffer_head * bh;
l 284     struct buffer_head * bh2;
l 285     int i, j, avefreei;
l 286     struct inode * inode;
l 287     int bitmap_nr;
l 288     struct ext2_group_desc * gdp;
l 289     struct ext2_group_desc * tmp;
l 290     struct ext2_super_block * es;
l 291
l 292     /* Cannot create files in a deleted directory */
l 293     if (!dir || !dir->i_nlink) {
l 294         *err = -EPERM;
l 295         return NULL;
l 296     }
l 297     /* initialize 一個inode的structure */
l 298     inode = get_empty_inode ();
l 299     if (!inode) {
l 300         *err = -ENOMEM;
l 301         return NULL;
l 302     }
```

Creating Inodes

ext2_new_inode()

```
l 304     sb = dir->i_sb;
l 305     inode->i_sb = sb;
l 306     inode->i_flags = 0;
l 307     lock_super (sb);
l /* The function tests and sets the value of the s_lock field
l    if necessary, suspends the current process until the flag becomes 0*/
l 308     es = sb->u.ext2_sb.s_es;
l 309 repeat:
l 310     gdp = NULL; i=0;
l 313     if (S_ISDIR(mode)) {
l 314         avefreei = le32_to_cpu(es->s_free_inodes_count) /
l 315             sb->u.ext2_sb.s_groups_count;
l 330         if (!gdp) {
l 331             for (j = 0; j < sb->u.ext2_sb.s_groups_count; j++) {
l 332                 tmp = ext2_get_group_desc (sb, j, &bh2); /* 取每人的gdp */
l 333                 if (tmp && /* 要有free且要> ave */
l 334                     le16_to_cpu(tmp->bg_free_inodes_count) &&
l 335                     le16_to_cpu(tmp->bg_free_inodes_count) >= avefreei) {
l 336                     if (!gdp || /* 比block */
l 337                         (le16_to_cpu(tmp->bg_free_blocks_count) >
l 338                          le16_to_cpu(gdp->bg_free_blocks_count)))
l /* i表示第幾個block group */
l 339                         i = j;
l 340                     gdp = tmp;}}}}}
```

Creating Inodes

ext2_new_inode()

```
| 346     else /* a */ /*不是directory*/
| 347     {
| 348         /*Try to place the inode in its parent directory */
| /* parent(dir)在哪個group*/
| 351         i = dir->u.ext2_i.i_block_group;
| 352         tmp = ext2_get_group_desc (sb, i, &bh2);
| /*if get到了此gdp且此group尚有free inode => 決定用此group*/
| 353         if (tmp && le16_to_cpu(tmp->bg_free_inodes_count))
| 354             gdp = tmp;
| 355         else
| 356         {
| 357             /*否則就二次方跳直到找到為此 */
| 361             for (j = 1; j < sb->u.ext2_sb.s_groups_count; j <= 1) {
| 362                 i += j; /* i initial 是dir的blk group*/
| 363                 if (i >= sb->u.ext2_sb.s_groups_count)
| 364                     i -= sb->u.ext2_sb.s_groups_count; /*超過最後從0繼續算*/
| 365                 tmp = ext2_get_group_desc (sb, i, &bh2);
| 366                 if (tmp &&
| 367                     le16_to_cpu(tmp->bg_free_inodes_count)) {
| 368                     gdp = tmp;
| 369                     break;
| 370                 }
|     }
| }
```

Creating Inodes

ext2_new_inode()

```
| /*假如還是沒找到就linear search任一個free inode*/
| 373         if (!gdp) {             /* b */
| 374             /*
| 375              * That failed: try linear search for a free inode
| 376              */
| 377             i = dir->u.ext2_i.i_block_group + 1;
| 378             for (j = 2; j < sb->u.ext2_sb.s_groups_count; j++) {
| 379                 if (++i >= sb->u.ext2_sb.s_groups_count)
| 380                     i = 0;
| 381                 tmp = ext2_get_group_desc (sb, i, &bh2);
| 382                 if (tmp &&
| 383                     le16_to_cpu(tmp->bg_free_inodes_count)) {
| 384                     gdp = tmp;
| 385                     break;
| 386                 }
| 387             }
| 388         }}
| 390 /* 還是找不到gdp => return NULL */
| 391         if (!gdp) {
| 392             unlock_super (sb);
| 393             iput(inode); /* trying to free free inode*/
| 394             return NULL;
| 395         }
```

Creating Inodes

ext2_new_inode()

```
|  /* load inode bitmap and return slot # */
|  396      bitmap_nr = load_inode_bitmap (sb, i);
|  397      if (bitmap_nr < 0) {
|  398          unlock_super (sb);
|  399          iput(inode);
|  400          *err = -EIO;
|  401          return NULL;
|  402      }
|  403          /* inode bitmap cache */
|  404      bh = sb->u.ext2_sb.s_inode_bitmap[bitmap_nr];
|  405      if ((j = ext2_find_first_zero_bit ((unsigned long *) bh->b_data,
|  406          EXT2_INODES_PER_GROUP(sb))) <
|  407          EXT2_INODES_PER_GROUP(sb)) {
|  408          if (ext2_set_bit (j, bh->b_data)) {
|  409              ext2_warning (sb, "ext2_new_inode",
|  410                  "bit already set for inode %d", j);
|  411              goto repeat;
|  412          }
|  413          mark_buffer_dirty(bh, 1);
|  if (sb->s_flags & MS_SYNCHRONOUS) {
|  /* This function can be used to request a number of buffers from a block
|  device. Currently the only restriction is that all buffers must belong to the same device */
|  415          ll_rw_block (WRITE, 1, &bh);
|  416          wait_on_buffer (bh);}

```

Creating Inodes

ext2_new_inode()

```
| 418     } else {
| /* if inode bitmap第一個等於0的位置沒小於bitmap的大小,可用的inode數又不是0時 */
| 419         if (le16_to_cpu(gdp->bg_free_inodes_count) != 0) {
| 420             ext2_error (sb, "ext2_new_inode",
| 421                 "Free inodes count corrupted in group %d",
| 422                 i);
| 423             unlock_super (sb);
| 424             iput (inode); /* trying to free free inode*/
| 425             return NULL;
| 426         }
| 427
| 428     }
| 429     j += i * EXT2_INODES_PER_GROUP(sb) + 1;
| 430     if (j < EXT2_FIRST_INO(sb) || j > le32_to_cpu(es->s_inodes_count)) {
| 431         ext2_error (sb, "ext2_new_inode",
| 432             "reserved inode or inode > inodes count - "
| 433             "block_group = %d,inode=%d", i, j);
| 434         unlock_super (sb);
| 435         iput (inode);
| 436         return NULL;
| 437     }
```

Creating Inodes

ext2_new_inode()

```
l 438     gdp->bg_free_inodes_count =
l 439         cpu_to_le16(le16_to_cpu(gdp->bg_free_inodes_count) - 1);
l 440     if (S_ISDIR(mode))
l 441         gdp->bg_used_dirs_count =
l 442             cpu_to_le16(le16_to_cpu(gdp->bg_used_dirs_count) + 1);
l 443     mark_buffer_dirty(bh2, 1);
l 444     es->s_free_inodes_count =
l 445         cpu_to_le32(le32_to_cpu(es->s_free_inodes_count) - 1);
l 446     mark_buffer_dirty(sb->u.ext2_sb.s_sbh, 1);
l 447     sb->s_dirt = 1;
l 448     inode->i_mode = mode;
l 449     inode->i_sb = sb;
l 450     inode->i_nlink = 1;
l 451     inode->i_dev = sb->s_dev;
l 452     inode->i_uid = current->fsuid;
l 453     if (test_opt (sb, GRPID))
l 454         inode->i_gid = dir->i_gid;
l 455     else if (dir->i_mode & S_ISGID) {
l 456         inode->i_gid = dir->i_gid;
l 457         if (S_ISDIR(mode))
l 458             mode |= S_ISGID;
l 459     } else
l 460         inode->i_gid = current->fsgid;
```

Creating Inodes

ext2_new_inode()

```
| .....  
| 462     inode->i_ino = j;  
| 464     inode->i_blocks = 0;  
| 465     inode->i_mtime = inode->i_atime = inode->i_ctime = CURRENT_TIME;  
| 466     inode->u.ext2_i.i_new_inode = 1;  
| .....  
| 489     mark_inode_dirty(inode);  
| 490  
| 491     unlock_super (sb);  
| 492     if(DQUOT_ALLOC_INODE(sb, inode)) {  
| 493         sb->dq_op->drop(inode);  
| 494         inode->i_nlink = 0;  
| 495         iput(inode);  
| 496         *err = -EDQUOT;  
| 497         return NULL;  
| 498     }  
| 499     ext2_debug ("allocating inode %lu\n", inode->i_ino);  
| 500  
| 501     *err = 0;  
|      /* return the new address of the new inode object */  
| 502     return inode;  
| 503 }
```

Creating Inodes

ext2_new_inode() è load_inode_bitmap ()

```
| 91 static int load_inode_bitmap (struct super_block * sb,  
| 92                               unsigned int block_group)  
| 93 {  
| 94     int i, j, retval = 0;  
| 95     unsigned long inode_bitmap_number;  
| 96     struct buffer_head * inode_bitmap;  
| 97  
  
| /* block group > 系統內的blk group數 => panic*/  
| 98     if (block_group >= sb->u.ext2_sb.s_groups_count)  
| 99         ext2_panic (sb, "load_inode_bitmap",  
| 100                    "block_group >= groups_count - "  
| 101                    "block_group = %d, groups_count = %lu",  
| 102                    block_group, sb->u.ext2_sb.s_groups_count);  
  
| /* if block group 是以存在cache 0中位置的, 就return 0*/  
| 103     if (sb->u.ext2_sb.s_loaded_inode_bitmaps > 0 &&  
| 104         sb->u.ext2_sb.s_inode_bitmap_number[0] == block_group &&  
| 105         sb->u.ext2_sb.s_inode_bitmap[0] != NULL)  
| 106         return 0;
```

Creating Inodes

ext2_new_inode() è load_inode_bitmap ()

```
| /* if the number of block groups in the Ext2 partition is less than or equal to
| EXT2_MAX_GROUP_LOADED, the index of the cache array position in which the bitmap is inserted
| always matches the block group index passed as the parameter to the load_inode_bitmap() function */
| 107     if (sb->u.ext2_sb.s_groups_count <= EXT2_MAX_GROUP_LOADED) {
| 108         if (sb->u.ext2_sb.s_inode_bitmap[block_group]) {
| 109             if (sb->u.ext2_sb.s_inode_bitmap_number[block_group] != block_group)
| 110                 ext2_panic (sb, "load_inode_bitmap",
| 111                     "block_group != inode_bitmap_number");
| 112             else
| 113                 return block_group;
| 114             } else {
| /* 還沒read進cache è read*/
| 115         retval = read_inode_bitmap (sb, block_group,
| 116             block_group);
| 117         if (retval < 0)
| 118             return retval;
| 119         return block_group;
| 120     }
| 121 }
```

Creating Inodes

ext2_new_inode() è load_inode_bitmap ()

```
|  /* LRU */
|  123      for (i = 0; i < sb->u.ext2_sb.s_loaded_inode_bitmaps &&
|  124          sb->u.ext2_sb.s_inode_bitmap_number[i] != block_group;
|  125          i++)
|  126          ;
|  127      if (i < sb->u.ext2_sb.s_loaded_inode_bitmaps &&
|  128          sb->u.ext2_sb.s_inode_bitmap_number[i] == block_group) {
|  129          inode_bitmap_number = sb->u.ext2_sb.s_inode_bitmap_number[i];
|  130          inode_bitmap = sb->u.ext2_sb.s_inode_bitmap[i];
|  /* 如果發理是在cache位置5上面，就把4->5，3->4，...最後再把新的放在0上面*/
|  131          for (j = i; j > 0; j--) {
|  132              sb->u.ext2_sb.s_inode_bitmap_number[j] =
|  133                  sb->u.ext2_sb.s_inode_bitmap_number[j - 1];
|  134              sb->u.ext2_sb.s_inode_bitmap[j] =
|  135                  sb->u.ext2_sb.s_inode_bitmap[j - 1];
|  136          }
|  137          sb->u.ext2_sb.s_inode_bitmap_number[0] = inode_bitmap_number;
|  138          sb->u.ext2_sb.s_inode_bitmap[0] = inode_bitmap;
|  139
|  140
```

Creating Inodes

ext2_new_inode()è load_inode_bitmap ()

```

|         /*
| 141         * There's still one special case here --- if inode_bitmap == 0
| 142         * then our last attempt to read the bitmap failed and we have
| 143         * just ended up caching that failure. Try again to read it.
| 144         */
| 145         if (!inode_bitmap)
| 146             retval = read_inode_bitmap (sb, block_group, 0);

|  /* if找得到就重排 cache，不然從disk read到cache先 */
| 148     } else {
| 149         if (sb->u.ext2_sb.s_loaded_inode_bitmaps < EXT2_MAX_GROUP_LOADED)
| 150             sb->u.ext2_sb.s_loaded_inode_bitmaps++;
| 151         else /* brelse : Release a buffer head */
| 152             brelse (sb->u.ext2_sb.s_inode_bitmap[EXT2_MAX_GROUP_LOADED - 1]);
| 153         for (j = sb->u.ext2_sb.s_loaded_inode_bitmaps - 1; j > 0; j--) {
| 154             sb->u.ext2_sb.s_inode_bitmap_number[j] =
| 155                 sb->u.ext2_sb.s_inode_bitmap_number[j - 1];
| 156             sb->u.ext2_sb.s_inode_bitmap[j] =
| 157                 sb->u.ext2_sb.s_inode_bitmap[j - 1];
| 158         }
| 159         retval = read_inode_bitmap (sb, block_group, 0);
| 160     }
| 161     return retval;
| 162 }
```

I Deleting Inodes (ext2_free_inode())

lock_super()	load_inode_bitmap()
clear_inode()	inc. bg_free_inodes_count
inc. s_free_inodes_count	ll_rw_block()
unlock_super()	

Data Blocks Addressing

(b:block size)

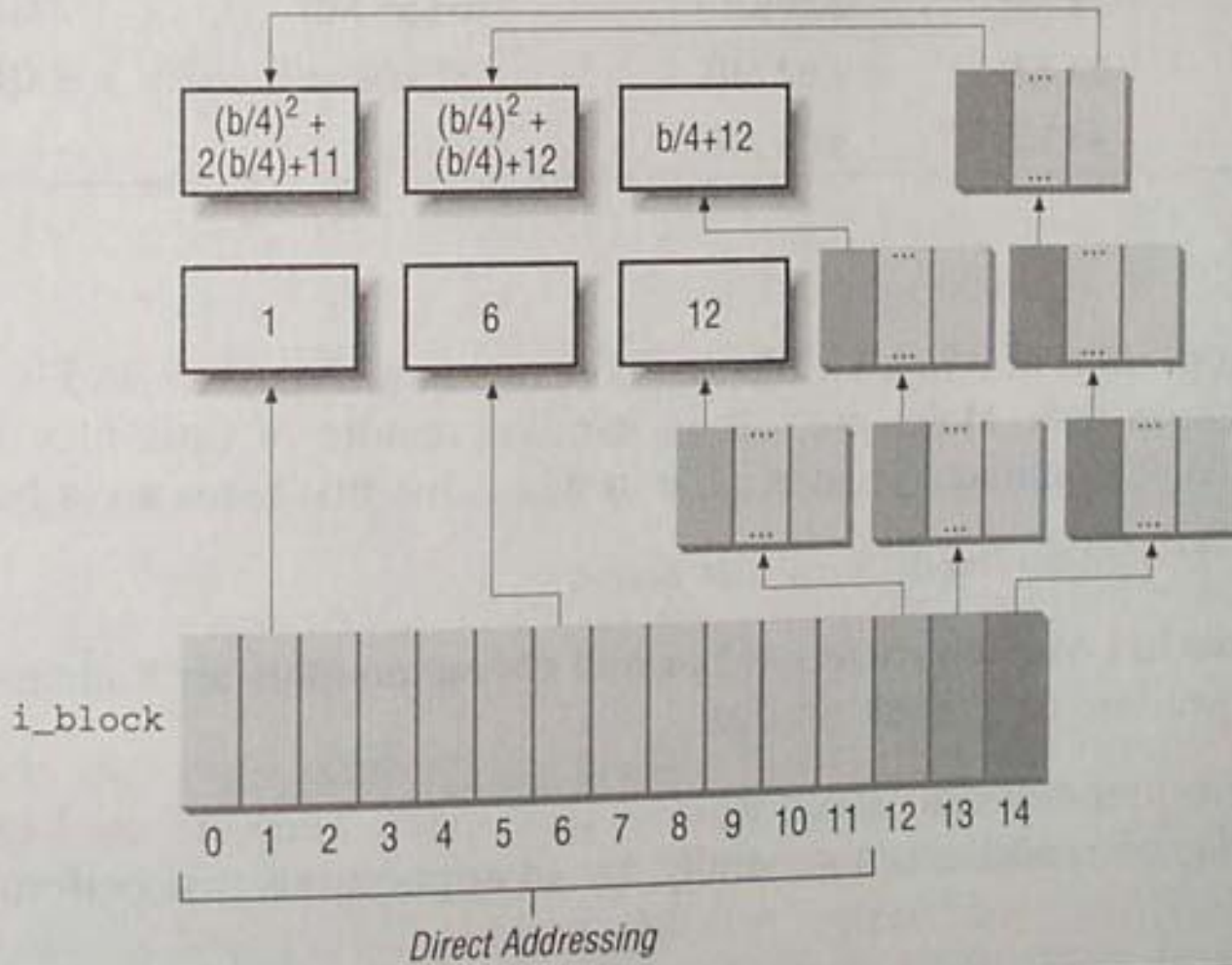


Figure 17-5. Data structures used to address the file's data blocks



I Allocating a Data Block

`ext2_getblk()`

`ext2_alloc_block()`

Allocating a Data Block

ext2_getblk()

```
| 367 struct buffer_head * ext2_getblk (struct inode * inode, long block,  
| 368                               int create, int * err)  
| 369 {  
| 370     struct buffer_head * bh;  
| 371     int offsets[4], *p;  
| 372     int depth = ext2_block_to_path(inode, block, offsets);  
| 373  
| 375     if (depth == 0)  
| 376         goto fail;  
| 377  
| 387     if (block == inode->u.ext2_i.i_next_alloc_block + 1) {  
| 388         inode->u.ext2_i.i_next_alloc_block++;  
| 389         inode->u.ext2_i.i_next_alloc_goal++;  
| 390     }  
| 391  
| 393     bh = inode_getblk (inode, *(p=offsets), create, block, err);  
| 394     while (--depth) {  
| 395         bh = block_getblk (inode, bh, *++p, create,  
| 396                             inode->i_sb->s_blocksize, block, err);  
| 397     }  
| 398     return bh;  
| 399 fail:  
| 400     return NULL;  
| 401 }
```

Allocating a Data Block

ext2_getblk() è ext2_block_to_path()

```
| 170 static int ext2_block_to_path(struct inode *inode, long i_block, int offsets[4])
| 171 {
| 172     int ptrs = EXT2_ADDR_PER_BLOCK(inode->i_sb); /* b/4 */
| 173     int ptrs_bits = EXT2_ADDR_PER_BLOCK_BITS(inode->i_sb);
| 174     const long direct_blocks = EXT2_NDIR_BLOCKS,
| 175             indirect_blocks = ptrs,
| 176             double_blocks = (1 << (ptrs_bits * 2)); /* (b/4)^2 */
| 177     int n = 0;
| 178
| 179     if (i_block < 0) {
| 180         ext2_warning (inode->i_sb, "ext2_block_to_path", "block < 0");
| 181     } else if (i_block < direct_blocks) {
| 182         offsets[n++] = i_block;
| 183     } else if ( (i_block -= direct_blocks) < indirect_blocks) {
| 184         offsets[n++] = EXT2_IND_BLOCK;
| 185         offsets[n++] = i_block;
```

Allocating a Data Block

ext2_getblk() è ext2_block_to_path()

```
I 186     } else if ((i_block -= indirect_blocks) < double_blocks) {
I 187         offsets[n++] = EXT2_DIND_BLOCK;
I 188         offsets[n++] = i_block >> ptrs_bits;           /* 取high bits */
I 189         offsets[n++] = i_block & (ptrs - 1);           /* 取low bits */
I 190     } else if (((i_block -= double_blocks) >> (ptrs_bits * 2)) < ptrs) {
I 191         offsets[n++] = EXT2_TIND_BLOCK;
I 192         offsets[n++] = i_block >> (ptrs_bits * 2);
I 193         offsets[n++] = (i_block >> ptrs_bits) & (ptrs - 1);
I 194         offsets[n++] = i_block & (ptrs - 1);
I 195     } else { /* 所給block太大 */
I 196         ext2_warning (inode->i_sb, "ext2_block_to_path", "block > big");
I 197     }
I 198     return n;
I 199 }
```

Allocating a Data Block

ext2_getblk() è inode_getblk()

```
I 227 static struct buffer_head * inode_getblk (struct inode * inode, int nr,
I 228                                         int create, int new_block, int * err)
I 229 {
I 230     u32 * p;
I 231     int tmp, goal = 0;
I 232     struct buffer_head * result;
I 233     int blocks = inode->i_sb->s_blocksize / 512;
I 235     p = inode->u.ext2_i.i_data + nr;
I 236 repeat:
I /* if the block being allocated and the previously allocated one have consecutive file block #, the goal is the
I logical block # of the previous block plus 1; it makes sense that consecutive blocks as seen by a program
I should be adjacent on disk */
I 250     if (inode->u.ext2_i.i_next_alloc_block == new_block) /* 剛才加的 */
I 251         goal = inode->u.ext2_i.i_next_alloc_goal;
I 255     if (!goal) {
I /* 選一個之前已allocate過的block */
I 256         for (tmp = nr - 1; tmp >= 0; tmp--) {
I 257             if (inode->u.ext2_i.i_data[tmp]) {
I 258                 goal = le32_to_cpu(inode->u.ext2_i.i_data[tmp]);
I 259                 break;
I 260             }
I 261         }
```

Allocating a Data Block

ext2_getblk() è inode_getblk()

```
| 262     if (!goal)
| /* 再沒有就找這個block group的第一個block */
| 263         goal = (inode->u.ext2_i.i_block_group *
| 264             EXT2_BLOCKS_PER_GROUP(inode->i_sb)) +
| 265             le32_to_cpu(inode->i_sb->u.ext2_sb.s_es->s_first_data_block);
| 266     }
| 267
| 268     ext2_debug ("goal = %d.\n", goal);
| 269     /* tmp == logical block # */
| 270     tmp = ext2_alloc_block (inode, goal, err);
| 271     if (!tmp)
| 272         return NULL;
| 273     result = getblk (inode->i_dev, tmp, inode->i_sb->s_blocksize);
| 274     *p = cpu_to_le32(tmp);
| 275     inode->u.ext2_i.i_next_alloc_block = new_block;
| 276     inode->u.ext2_i.i_next_alloc_goal = tmp;
| 277     inode->i_ctime = CURRENT_TIME;
| 278     inode->i_blocks += blocks;
| 279     return result;
| 280 }
| 281 }
```

Allocating a Data Block

`ext2_getblk()` è `inode_getblk()` è `ext2_alloc_block()` è `ext2_new_block`

```
360 int ext2_new_block (const struct inode * inode, unsigned long goal,
361                    u32 * prealloc_count, u32 * prealloc_block, int * err)
362 {
    .....
394 repeat:
| 395     /*
| 396     * First, test whether the goal block is free.
| 397     */
| /* 假如goal out of range, 把goal設在第一個 */
| 398     if (goal < le32_to_cpu(es->s_first_data_block) ||
| 399         goal >= le32_to_cpu(es->s_blocks_count))
| 400         goal = le32_to_cpu(es->s_first_data_block);
| /* get gdp, i è goal在那個block group */
| 401     i = (goal - le32_to_cpu(es->s_first_data_block)) / EXT2_BLOCKS_PER_GROUP(sb);
| 402     gdp = ext2_get_group_desc (sb, i, &bh2);
| 403     if (!gdp)
| 404         goto io_error;
| /* j è goal在block group的那一個block */
| 406     if (le16_to_cpu(gdp->bg_free_blocks_count) > 0) {
| 407         j = ((goal - le32_to_cpu(es->s_first_data_block)) %
EXT2_BLOCKS_PER_GROUP(sb));
```

Allocating a Data Block

ext2_getblk() è inode_getblk()è ext2_alloc_block()è ext2_new_block

/* 把bitmap load 出*/

```
412     bitmap_nr = load_block_bitmap (sb, i);
413     if (bitmap_nr < 0)
414         goto io_error;
415
416     bh = sb->u.ext2_sb.s_block_bitmap[bitmap_nr];
```

/* 假如goal指的bit沒在用就 goal hit了*/

/* bh->b_data指向此buffer的地址 */

/* if the goal is free, allocates it */

```
420     if (!ext2_test_bit(j, bh->b_data)) {
425         goto got_block;
426     }

427     if (j) { /* j è goal在block group的那一個block */
436         int end_goal = (j + 63) & ~63;
437         j = ext2_find_next_zero_bit(bh->b_data, end_goal, j);
438         if (j < end_goal)
439             goto got_block;
440     }
```

Allocating a Data Block

`ext2_getblk()` è `inode_getblk()` è `ext2_alloc_block()` è `ext2_new_block`

```
/* 還是沒有的話就找bitmap有一個byte free的,再沒有就找bit free的
從現在這個group開始找，接下來循環找所有group */
453         p = ((char *) bh->b_data) + (j >> 3);
/* void * memscan(void * addr, int c, size_t size) */
/* find the first occurrence of byte 'c', or 1 past the area if none */
/* 從j 的那個位置開始找，第一個0的位址放到k，k要再乘8，因為我們是bit search */
454         r = memscan(p, 0, (EXT2_BLOCKS_PER_GROUP(sb) - j + 7) >> 3);
455         k = (r - ((char *) bh->b_data)) << 3;
456         if (k < EXT2_BLOCKS_PER_GROUP(sb)) {
457             j = k;
/* 最好要byte free */
458             goto search_back;
459         }
/* 沒有byte free找bit free */
461         k = ext2_find_next_zero_bit ((unsigned long *) bh->b_data,
462                                     EXT2_BLOCKS_PER_GROUP(sb),
463                                     j);
464         if (k < EXT2_BLOCKS_PER_GROUP(sb)) {
465             j = k;
466             goto got_block;
467         }
468     }
```

Allocating a Data Block

`ext2_getblk()` è `inode_getblk()` è `ext2_alloc_block()` è `ext2_new_block`

```
/* 找第一個有free block的block group */
476     for (k = 0; k < sb->u.ext2_sb.s_groups_count; k++) {
477         i++;
478         if (i >= sb->u.ext2_sb.s_groups_count)
479             i = 0;
480         gdp = ext2_get_group_desc (sb, i, &bh2);
481         if (!gdp) {
482             *err = -EIO;
483             unlock_super (sb);
484             return 0;
485         }
486         if (le16_to_cpu(gdp->bg_free_blocks_count) > 0)
487             break;
488     }
/* 找不到半個free的就return 0 */
489     if (k >= sb->u.ext2_sb.s_groups_count) {
490         unlock_super (sb);
491         return 0;
492     }
493     bitmap_nr = load_block_bitmap (sb, i);
494     if (bitmap_nr < 0)
495         goto io_error;
```

Allocating a Data Block

`ext2_getblk()` è `inode_getblk()` è `ext2_alloc_block()` è `ext2_new_block`

/ 找到第一個有free block的block group後，和剛才一樣的動作找free block */*

```
497     bh = sb->u.ext2_sb.s_block_bitmap[bitmap_nr];
498     r = memscan(bh->b_data, 0, EXT2_BLOCKS_PER_GROUP(sb) >> 3);
499     j = (r - bh->b_data) << 3;
500     if (j < EXT2_BLOCKS_PER_GROUP(sb))
501         goto search_back;
502     else
503         j = ext2_find_first_zero_bit ((unsigned long *) bh->b_data,
504                                     EXT2_BLOCKS_PER_GROUP(sb));
```

/ j太大 error */*

```
505     if (j >= EXT2_BLOCKS_PER_GROUP(sb)) {
506         ext2_error (sb, "ext2_new_block",
507                  "Free blocks count corrupted for block group %d", i);
508         unlock_super (sb);
509         return 0;
510     }
```

Allocating a Data Block

`ext2_getblk()` è `inode_getblk()` è `ext2_alloc_block()` è `ext2_new_block`

```
512 search_back:
```

```
513     /*
```

```
514     * We have succeeded in finding a free byte in the block
```

```
515     * bitmap. Now search backwards up to 7 bits to find the
```

```
516     * start of this group of free blocks.
```

```
517     */
```

```
518     for (k = 0; k < 7 && j > 0 && !ext2_test_bit (j - 1, bh->b_data); k++, j--);
```

```
519
```

```
520 got_block:
```

```
521 tmp = j + i * EXT2_BLOCKS_PER_GROUP(sb) + le32_to_cpu(es->s_first_data_block);
```

```
522     /*
```

```
523     * Do block preallocation now if required.
```

```
524     */
```

```
525 #ifdef EXT2_PREALLOCATE
```

```
526     if (prealloc_count && !*prealloc_count) {
```

```
527         int    prealloc_goal;
```

```
528         unsigned long  next_block = tmp + 1;
```

```
529         /* 0是用default 8個，其他則幾個是幾個 */
```

```
530         prealloc_goal = es->s_prealloc_blocks ?
```

```
531             es->s_prealloc_blocks : EXT2_DEFAULT_PREALLOC_BLOCKS;
```

```
532
```

Allocating a Data Block

`ext2_getblk()` è `inode_getblk()` è `ext2_alloc_block()` è `ext2_new_block`

```
563     *prealloc_block = next_block;
      for (k = 1; k < prealloc_goal && (j + k) < EXT2_BLOCKS_PER_GROUP(sb); k++, next_block++) {
566         /* 1==NO_QUOTA */
567         if (DQUOT_PREALLOC_BLOCK(sb, inode, 1))
568             break;
574         (*prealloc_count)++;
575     }
/* preallocate 了幾個就要減去幾個free_count */
576     gdp->bg_free_blocks_count =
577         cpu_to_le16(le16_to_cpu(gdp->bg_free_blocks_count) -
578             (k - 1));
579     es->s_free_blocks_count =
580         cpu_to_le32(le32_to_cpu(es->s_free_blocks_count) -
581             (k - 1));
582     ext2_debug ("Preallocated a further %lu bits.\n",
583         (k - 1));
584 }
624 return tmp;
625 }
```

I Releasing a Data Block

1. Lock_super()
2. Gets the block bitmap of the block group including the block to be released
3. Clears the bit in the block bitmap, and marks the buffer as dirty
4. Inc. bg_free_blocks_count
5. Inc. s_free_blocks_count
6. Ll_rw_block()
7. Unlock_super()

Writing an Ext2 Regular file

ext2_file_write ()

```
154 static ssize_t ext2_file_write (struct file * filp, const char * buf,
155                                 size_t count, loff_t *ppos)
156 {
157     struct inode * inode = filp->f_dentry->d_inode;
158     off_t pos;
159     long block;
160     int offset;
161     int written, c;
162     struct buffer_head * bh, *bufferlist[NBUF];
163     struct super_block * sb;
164     int err;
165     int i,buffercount,write_error, new_buffer;
166     unsigned long limit;
167
168     /* POSIX: mtime/ctime may not change for 0 count */
169     if (!count)
170         return 0;
171     /* This makes the bounds-checking arithmetic later on much more
172      * sane. */
173     if (((signed) count) < 0)
174         return -EINVAL;
```

Writing an Ext2 Regular file

ext2_file_write ()

```
176     write_error = buffercount = 0;
177     if (!inode) {
178         printk("ext2_file_write: inode = NULL\n");
179         return -EINVAL;
180     }
181     sb = inode->i_sb;
182     if (sb->s_flags & MS_RDONLY) /* read only */
183         /*
184          * This fs has been automatically remounted ro because of errors
185          */
186         return -ENOSPC;
187
188     if (!S_ISREG(inode->i_mode)) { /* is a regular File? */
189         ext2_warning (sb, "ext2_file_write", "mode = %07o",
190                     inode->i_mode);
191         return -EINVAL;
192     }
193     /* removes any superuser privilege from the file (to guard against tampering with set-uid programs) */
194     remove_suid(inode);
```

Writing an Ext2 Regular file

ext2_file_write ()

```
/* if the file has been opened with the O_APPEND flag set, sets the file offset where data must be written to the end of the file */
```

```
195     if (filp->f_flags & O_APPEND)
196         pos = inode->i_size;
197     else {
198         pos = *ppos;
199         if (pos != *ppos)
200             return -EINVAL;
201     }
```

```
/* if the file has been opened in synchronous mode (O_SYNC flag set), sets i_osync field in the ext2_inode_info of the disk inode to 1, this flag is tested when a data block is allocated for the file, so that the kernel can synchronously update the inode on disk as soon as it is modified */
```

```
264     if (filp->f_flags & O_SYNC)
265         inode->u.ext2_i.i_osync++;
    /* 在第幾個blocks==pos / block size */
266     block = pos >> EXT2_BLOCK_SIZE_BITS(sb);
    /* 找出pos在此block的offset */
267     offset = pos & (sb->s_blocksize - 1);
268     c = sb->s_blocksize - offset;
    /* if (c > count) c=count;
    else (c用盡==此block滿), count -c 再alloc新blk */
269     written = 0;
```

Writing an Ext2 Regular file

ext2_file_write ()

```
/* for each block to be written, performs the following substeps */
268     do {                               /* to 353 */
/* to get the data block on the disk, allocating it when necessary */
271         bh = ext2_getblk (inode, block, 1, &err);
272         if (!bh) {
273             if (!written)
274                 written = err;
275             break;
276         }
277         if (c > count)
278             c = count;
291         new_buffer = (!buffer_uptodate(bh) && !buffer_locked(bh) &&
292             c == sb->s_blocksize);
293
294         if (new_buffer) {
295             set_bit(BH_Lock, &bh->b_state);
/* copy from user回傳沒co到的#, always 0 */
296             c -= copy_from_user (bh->b_data + offset, buf, c);
```

Writing an Ext2 Regular file

ext2_file_write ()

```
297         if (c != sb->s_blocksize) {
298             c = 0;
299             unlock_buffer(bh);
300             /* Release a buffer head */
301             brelse(bh);
302             if (!written)
303                 written = -EFAULT;
304             break;
305         }
306         mark_buffer_uptodate(bh, 1);
307         unlock_buffer(bh);
308     } else {
```

/ if the block has to be partially rewritten and the buffer is not up-to-date, invokes ll_rw_block() and waits until the read operation terminates */*

```
308         if (!buffer_uptodate(bh)) {
309             ll_rw_block (READ, 1, &bh);
310             wait_on_buffer (bh);
311             if (!buffer_uptodate(bh)) {
312                 brelse (bh);
313                 if (!written)
314                     written = -EIO;
315                 break;}}
```

Writing an Ext2 Regular file

ext2_file_write ()

```
        c -= copy_from_user (bh->b_data + offset, buf, c);
319     }
320     if (!c) {
321         brelse(bh);
322         if (!written)
323             written = -EFAULT;
324         break;
325     }
326     mark_buffer_dirty(bh, 0);
/* synchronize the contents of the page cache with that of the buffer cache */
327     update_vm_cache_conditional(inode, pos, bh->b_data + offset, c,
328                               (unsigned long) buf);

/* 寫了c個到buffer */
329     pos += c;
330     written += c;
331     buf += c;
332     count -= c;
333
334     if (filp->f_flags & O_SYNC)
335         bufferlist[buffercount++] = bh;
```

Writing an Ext2 Regular file

ext2_file_write ()

```
336         else
337             brelse(bh);
          /* 將 buffer write 到 file */
338         if (buffercount == NBUF){
339             ll_rw_block(WRITE, buffercount, bufferlist);
340             for(i=0; i<buffercount; i++){
341                 wait_on_buffer(bufferlist[i]);
342                 if (!buffer_uptodate(bufferlist[i]))
343                     write_error=1;
344                 brelse(bufferlist[i]);
345             }
346             buffercount=0;
347         }
348         if(write_error)
349             break;
350         block++;
351         offset = 0;
352         c = sb->s_blocksize;
353     } while (count);
```

/* 270的do的結束 */

Writing an Ext2 Regular file

ext2_file_write ()

```
354     if ( buffercount ){
355         ll_rw_block(WRITE, buffercount, bufferlist);
356         for(i=0; i<buffercount; i++){
357             wait_on_buffer(bufferlist[i]);
358             if (!buffer_uptodate(bufferlist[i]))
359                 write_error=1;
360             brelse(bufferlist[i]);
361         }
362     }
363     if (pos > inode->i_size)
364         inode->i_size = pos;
365     if (filp->f_flags & O_SYNC)
366         inode->u.ext2_i.i_osync--;
367     inode->i_ctime = inode->i_mtime = CURRENT_TIME;
368     *ppos = pos;
369     mark_inode_dirty(inode);
370     /* return # of bytes written into the file */
371     return written;
371 }
```

Reference

BOOKS

- | Understanding the linux kernel – O'RELLY
- | Operating system concepts – ADDISON WESLEY

WEB SITES

- | Study-area
- | lxr.linux.no è linux kernel 2.2.20





