Note that the conversion is valid only if everything counts from 0. If some of the items do not count from 0, they can not be used in the conversion formula directly. The reader may try to figure out how to handle such cases in general. For ease of reference, we shall refer to the conversion method as the Mailman's algorithm.

2.7.1 Applications of Mailman's Algorithm

1. Test, Set and Clear bits in C: In standard C programs, the smallest addressable unit is a char or byte. It is often necessary to manipulate bits in a bitmap, which is a sequence of bits. Consider char buf[1024], which has 1024 bytes, denoted by buf[i], i = 0, 1,…., 1023. It also has 8192 bits numbered 0,1,2,….8191. Given a bit number BIT, e.g. 1234, which byte i contains the bit, and which bit j is it in that byte? Solution:

```
i = BIT / 8;   j = BIT % 8;   // 8 = number of bits in a byte.
```

This allows us to combine the Mailman's algorithm with bit masking to do the following bit operations in C.

```
.TST a bit for 1 or 0 :  if (buf[i] &   (1 << j))
.SET a bit to 1       :      buf[i] |=  (1 << j);
.CLR a bit to 0       :      buf[i] &= ~(1 << j);
```

It is noted that some C compilers allow specifying bits in a structure, as in

```
struct bits{
   unsigned int bit0      :  1;  // bit0 field is a single bit
   unsigned int bit123    :  3;  // bit123 field is a range of 3 bits
   unsigned int otherbits : 27;  // other bits field has 27 bits
   unsigned int bit31     :  1;  // bit31 is the highest bit
}var;
```

The structure defines var as an unsigned 32-bit integer with individual bits or ranges of bits. Then, var.bit0 = 0; assigns 1 to bit 0, and var.bit123 = 5; assigns 101 to bits 1 to 3, etc. However, the generated code still relies on the Mailman's algorithm and bit masking to access the individual bits. The Mailman's algorithm allows us to manipulate bits in a bitmap directly without defining complex C structures.

2. Convert INODE number to inode position on disk: In an EXT2 file system, each file has a unique INODE structure. On the file system disk, inodes begin in the inode_table block. Each disk block contains

```
INODES_PER_BLOCK = BLOCK_SIZE/sizeof(INODE)
```

inodes. Each inode has a unique inode number, ino = 1, 2,….., counted linearly from 1. Given an ino, e.g. 1234, determine which disk block contains the inode and which inode is it in that block? We need to know the disk block number because read/write a real disk is by blocks. Solution:

```
block = (ino - 1) / INODES_PER_BLOCK + inode_table;
inode = (ino - 1) % INODES_PER_BLOCK;
```

Similarly, converting double and triple indirect logical block numbers to physical block numbers in an EXT2 file system also depends on the Mailman's algorithm.

3. Convert linear disk block number to CHS = (cylinder, head, sector) format: Floppy disk and old hard disk use CHS addressing but file systems always use linear block addressing. The algorithm can be used to convert a disk block number to CHS when calling BIOS INT13.

2.8 EXT2 File System

For many years, Linux used EXT2 (Card et al.) as the default file system. EXT3 [ETX3] is an extension of EXT2. The main addition in EXT3 is a journal file, which records changes made to the file system in a journal log. The log allows for quicker recovery from errors in case of a file system crash. An EXT3 file system with no error is identical to an EXT2 file system. The newest extension of EXT3 is EXT4 (Cao et al. 2007). The major change in EXT4 is in the allocation of disk blocks. In EXT4, block numbers are 48 bits. Instead of discrete disk blocks, EXT4 allocates contiguous ranges of disk blocks, called extents. EXT4 file system will be covered in Chap. 3 when we develop booting programs for hard disks containing EXT4 file systems. MTX is a small operating system intended mainly for teaching. Large file storage capacity is not the design goal. Principles of file system design and implementation, with an emphasis on simplicity and compatibility with Linux, are the major focal points. For these reasons, MTX uses ETX2 as the file system. Support for other file systems, e.g. DOS and iso9660 (ECMA-119 1987), are not implemented in the MTX kernel. If needed, they can be implemented as user-level utility programs.

2.8.1 EXT2 File System Data Structures

Under MTX, the command, mkfs device nblocks [ninodes], creates an EXT2 file system on a specified device with nblocks blocks of 1 KB block size. The device can be either a real device or a virtual disk file. If the number of ninodes is not specified, the default number of inodes is nblocks/4. The resulting file system is totally compatible with the EXT2 file system of Linux. As a specific example, mkfs /dev/fd0

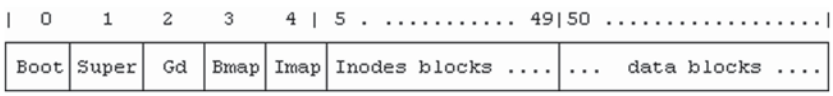

Fig. 2.17 Simple EXT2 file system layout

1440 makes an EXT2 file system on a 1.44 MB floppy disk with 1440 blocks and 360 inodes. Instead of a real device, the target can also be a disk image file. The layout of such an EXT2 file system is shown in Fig. 2.17.

For ease of discussion, we shall assume this basic file system layout first. Whenever appropriate, we point out the variations, including those in large EXT2/3 FS on hard disks. The following briefly explains the contents of the disk blocks.

Block#0: Boot Block: B0 is the boot block, which is not used by the file system. It contains a booter program for booting up an OS from the disk.

Block#1: Superblock: (at byte offset 1024 in hard disk partitions): B1 is the superblock, which contains information about the entire file system. Some of the important fields of the superblock structure are shown below.

```
struct ext2_super_block {
  u32  s_inodes_count;         // total number of inodes
  u32  s_blocks_count;         // total number of blocks
  u32  s_r_blocks_count;
  u32  s_free_blocks_count;    // current number of free blocks
  u32  s_free_inodes_count;    // current number of free inodes
  u32  s_first_data_block;     // first data block: 1 for FD, 0 for HD
  u32  s_log_block_size;       // 0 for 1KB block, 2 for 4KB block
  u32  s_log_frag_size;        // not used
  u32  s_blocks_per_group;     // number of blocks per group
  u32  s_frags_per_group;      // not used
  u32  s_inodes_per_group;
  u32  s_mtime, s_wtime;
  u16  s_mnt_count;            // number of times mounted
  u16  s_max_mnt_count;        // mount limit
  u16  s_magic;                // 0xEF53
  // MORE non-essential fileds,
  u16 s_inode_size=256 bytes for EXT4
};
```

Block#2: Group Descriptor Block (in s_first_data_block +1 on hard disk): EXT2 divides disk blocks into groups. Each group contains 8192 (32 K on HD) blocks. Each group is described by a group descriptor structure.

```
struct ext2_group_desc
{
  u32  bg_block_bitmap;        // Bmap block number
  u32  bg_inode_bitmap;        // Imap block number
  u32  bg_inode_table;         // Inodes begin block number
  u16  bg_free_blocks_count;   // THESE are OBVIOUS
  u16  bg_free_inodes_count;
  u16  bg_used_dirs_count;
  u16  bg_pad;                 // ignore these
  u32  bg_reserved[3];
};
```

Since a FD has only 1440 blocks, B2 contains only 1 group descriptor. The rest are 0's. On a hard disk with a large number of groups, group descriptors may span many blocks. The most important fields in a group descriptor are bg_block_bitmap, bg_inode_bitmap and bg_inode_table, which point to the group's blocks bitmap, inodes bitmap and inodes start block, respectively. Here, we assume the bitmaps are in blocks 3 and 4, and inodes start from block 5.

Block#3: Block Bitmap (Bmap): (bg_block_bitmap): A bitmap is a sequence of bits used to represent some kind of items, e.g. disk blocks or inodes. A bitmap is used to allocate and deallocate items. In a bitmap, a 0 bit means the corresponding item is FREE, and a 1 bit means the corresponding item is IN_USE. A FD has 1440 blocks but block#0 is not used by the file system. So the Bmap has only 1439 valid bits. Invalid bits are treated as IN_USE and set to 1's.

Block#4: Inode Bitmap (Imap) (bg_inode_bitmap): An inode is a data structure used to represent a file. An EXT2 file system is created with a finite number of inodes. The status of each inode is represented by a bit in the Imap in B4. In an EXT2 FS, the first 10 inodes are reserved. So the Imap of an empty EXT2 FS starts with ten 1's, followed by 0's. Invalid bits are again set to 1's.

Block#5: Inodes (begin) Block (bg_inode_table): Every file is represented by a unique inode structure of 128 (256 in EXT4) bytes. The essential inode fields are listed below.

```
struct ext2_inode {
  u16  i_mode;          // 16 bits = |tttt|ugs|rwx|rwx|rwx|
  u16  i_uid;           // owner uid
  u32  i_size;          // file size in bytes
  u32  i_atime;         // time fields in seconds
  u32  i_ctime;         // since 00:00:00,1-1-1970
  u32  i_mtime;
  u32  i_dtime;
  u16  i_gid;           // group ID
  u16  i_links_count;   // hard-link count
  u32  i_blocks;        // number of 512-byte sectors
  u32  i_flags;         // IGNORE
  u32  i_reserved1;     // IGNORE
  u32  i_block[15];     // See details below
  u32  i_pad[7];
}
```

In the inode structure, i_mode specifies the file's type, usage and permissions. For example, the leading 4 bits = 1000 for REG file, 0100 for DIR, etc. The last 9 bits are the rwx permission bits for file protection. The i_block[15] array contains disk blocks of a file, which are

Direct blocks: i_block[0] to i_block[11], which point to direct disk blocks.

Indirect blocks: i_block[12] points to a disk block, which contains 256 block numbers, each of which points to a disk block.

Double Indirect blocks: i_block[13] points to a block, which points to 256 blocks, each of which points to 256 disk blocks.

Triple Indirect blocks: i_block[14] is the triple-indirect block. We may ignore this for "small" EXT2 FS.

The inode size (128 or 256) divides block size (1KB or 4KB) evenly, so that every inode block contains an integral number of inodes. In the simple EXT2 file system, the number of inode blocks is equal to ninodes/8. For example, if the number of inodes is 360, which needs 45 blocks, the inode blocks include B5 to B49. Each inode has a unique inode number, which is the inode's position in the inode blocks plus 1. Note that inode positions count from 0, but inode numbers count from 1. A 0 inode number means no inode. The root directory's inode number is 2.

Data Blocks: Immediately after the inode blocks are data blocks. Assuming 360 inodes, the first real data block is B50, which is i_block[0] of the root directory /.

EXT2 Directory Entries: A directory contains dir_entry structures, in which the name field contains 1 to 255 chars. So the dir_entry's rec_len also varies.

```
struct ext2_dir_entry_2 {
   u32 inode;                // inode number; count from 1, NOT 0
   u16 rec_len;              // this entry's length in bytes
   u8  name_len;             // name length in bytes
   u8  file_type;            // not used
   char name[EXT2_NAME_LEN]; // name: 1  -255 chars, no NULL byte
};
```

2.8.2 Traverse EXT2 File System Tree

Given an EXT2 file system and the pathname of a file, e.g. /a/b/c, the problem is how to find the file. To find a file amounts to finding its inode. The algorithm is as follows.

1. Read in the superblock, which is at the byte offset 1024. Check the magic number s_magic (0xEF53) to verify it's indeed an EXT2 FS.
2. Read in the group descriptor block (1 + s_first_data_block) to access the group 0 descriptor. From the group descriptor's bg_inode_table entry, find the inodes begin block number, call it the InodesBeginBlock.
3. Read in InodeBeginBlock to get the inode of /, which is INODE #2.
4. Tokenize the pathname into component strings and let the number of components be n. For example, if pathname = /a/b/c, the component strings are "a", "b", "c", with $n = 3$. Denote the components by name[0], name[1],..,name[n-1].
5. Start from the root INODE in (3), search for name[0] in its data block(s). For simplicity, we may assume that the number of entries in a DIR is small, so that a DIR inode only has 12 direct data blocks. With this assumption, it suffices to search the 12 direct blocks for name[0]. Each data block of a DIR INODE contains dir_entry structures of the form

[ino rlen nlen NAME] [ino rlen nlen NAME] .....

where NAME is a sequence of nlen chars (without a terminating NULL char). For each data block, read the block into memory and use a dir_entry *dp to point at the loaded data block. Then use nlen to extract NAME as a string and compare it with name[0]. If they do not match, step to the next dir_entry by

```
dp = (dir_entry *)((char *)dp + dp->rlen);
```

and continue the search. If name[0] exists, we can find its dir_entry and hence its inode number.

6. Use the inode number, ino, to locate the corresponding INODE. Recall that ino counts from 1. Use the Mailman's algorithm to compute the disk block containing the INODE and its offset in that block.

```
blk    = (ino - 1) / INODES_PER_BLOCK  + InodesBeginBlock;
offset = (ino - 1) % INODES_PER_BLOCK;
```

Then read in the INODE of /a, from which we can determine whether it's a DIR. If /a is not a DIR, there can't be /a/b, so the search fails. If it's a DIR and there are more components to search, continue for the next component name[1]. The problem now becomes: search for name[1] in the INODE of /a, which is exactly the same as that of Step (5).

7. Since Steps 5–6 will be repeated n times, it's better to write a search function

```
u32 search(INODE *inodePtr, char *name)
{
    // search for name in the data blocks of this INODE
    // if found, return its ino; else return 0
}
```

Then all we have to do is to call search() n times, as sketched below.

```
Assume: n,  name[0], ...., name[n-1] are globals
INODE *ip points at INODE of /
for (i=0; i<n; i++){
     ino = search(ip, name[i])
     if (!ino){ // can't find name[i], exit;}
     use ino to read in INODE and let ip point to INODE
}
```

If the search loop ends successfully, ip must point at the INODE of pathname. Traversing large EXT2 FS with many groups is similar, which will be shown in Chap. 3 when we discuss booting from hard disk partitions.