EXT2 File System

11

Abstract

This chapter covers EXT2 file system. The goal of this chapter is to lead the reader to implement a complete EXT2 file system that is totally Linux compatible. The premise is that if the reader understands one file system well, it should be easy to adapt to any other file systems. It first describes the historic role of EXT2 file system in Linux and the current status of EXT3/EXT4 file systems. It uses programming examples to show the various EXT2 data structures and how to traverse the EXT2 files system tree. Then it shows how to implement an EXT2 file system which supports all file operations as in the Linux kernel. It shows how to build a base file system by mount_root from a virtual disk. Then it divides the file system implementation into 3 levels. Level-1 expands the base file system to implement the file system tree. Level-2 implements read/write operations of file contents. Level-3 implements mount/umount of file systems and file protection. In each level, it describes the algorithms of the file system functions and demonstrates their implementations by programming examples. Each level is cumulated by a programming project. The final project is to integrate all the programming examples and exercises into a fully functional file system.

11.1 EXT2 File System

For many years, Linux used EXT2 (Card et al. 1995) as the default file system. EXT3 (ETX3, 2014) 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. Other than these minor changes, the file system structure and file operations remain the same. The purpose of this book is to teach the principles of file systems. Large file storage capacity is not the primary 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, we shall use ETX2 as the file system. The goal of this chapter is to lead the reader to implement a complete EXT2

K. C. Wang, *Systems Programming in Unix/Linux*, https://doi.org/10.1007/978-3-319-92429-8_11

file system that is totally Linux compatible. The premise is that if the reader understands one file system well, it should be relatively easy to adapt to any other file systems.

11.2 EXT2 File System Data Structures

11.2.1 Create Virtual Disk by mkfs

Under Linux, the command

```
mke2fs [-b blksize -N ninodes] device nblocks
```

creates an EXT2 file system on a device with nblocks blocks of blksize bytes per block and ninodes inodes. The device can be either a real device or a virtual disk file. If blksize is not specified, the default block size is 1 KB. If ninoides is not specified, mke2fs will compute a default ninodes number based on nblocks. The resulting EXT2 file system is ready for use in Linux. As a specific example, the following commands

```
dd if=/dev/zero of=vdisk bs=1024 count=1440
mke2fs vdisk 1440
```

creates an EXT2 files system on a virtual disk file named vdisk with 1440 blocks of 1 KB block size.

11.2.2 Virtual Disk Layout

The layout of such an EXT2 file system is shown in Fig. 11.1.

To begin with, we shall assume this basic file system layout first. Whenever appropriate, we point out the variations, including those in large EXT2/3 file systems 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 is used to contain a booter program for booting up an operating system from the disk.

11.2.3 Superblock

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.

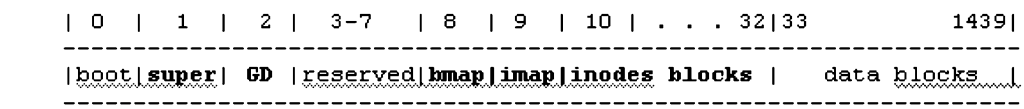


Fig. 11.1 Simple EXT2 file system layout

```
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 */
   u32  s_log_cluster_size;     /* Allocation cluster size */
   u32  s_blocks_per_group;     /* # Blocks per group */
   u32  s_clusters_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 */
   // more non-essential fields
   u16  s_inode_size;           /* size of inode structure */
}
```

The meanings of most superblock fields are obvious. Only a few fields deserve more explanation.

s_first_data_block = 0 for 4KB block size and 1 for 1KB block size. It is used to determine the start block of group descriptors, which is s_first_data_block + 1.

s_log_block_size determines the file block size, which is 1KB*(2**s_log_block_size), e.g.. 0 for 1KB block size, 1 for 2KB block size and 2 for 4KB block size, etc. The most often used block size is 1KB for small file systems and 4KB for large file systems.

s_mnt_count= number of times the file system has been mounted. When the mount count reaches the **max_mnt_count**, a fsck session is forced to check the file system for consistency.

s_magic is the magic number which identifies the file system type. For EXT2/3/4 files systems, the magic number is **0xEF53**.

11.2.4 Group Descriptors

Block#2: Group Descriptor Block (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 virtual floppy disk (FD) has only 1440 blocks, B2 contains only 1 group descriptor. The rest are 0's. On hard disks 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. For the Linux formatted EXT2 file system, blocks 3 to 7 are reserved. So bmap=8, imap=9 and inode_table = 10.

11.2.5 Block and Inode Bitmaps

Block#8: 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#9: 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 B9. 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.

11.2.6 Inodes

Block#10: 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];        // for inode size = 128 bytes
 }
```

In the inode structure, i_mode is a u16 or 2-byte unsigned integer.

```
           | 4  | 3 |   9     |
 i_mode = |tttt|ugs|rwxrwxrwx|
```

In the i_mode field, the leading 4 bits specify the file type, e.g. tttt=1000 for REG file, 0100 for DIR, etc. The next 3 bits ugs indicate the file's special usage. The last 9 bits are the rwx permission bits for file protection.

The i_size field is the file size in bytes. The various time fields are number of seconds elapsed since 0 hr 0 min, 0 s of January 1, 1970. So each time filed is a very large unsigned integer. They can be converted to calendar form by the library function

```
char *ctime(&time_field)
```

which takes a pointer to a time field and returns a string in calendar form. For example,

```
printf("%s", ctime(&inode.i_atime); // note: pass & of time field prints i_atime
in calendar form.
```

The i_block[15] array contains pointers to 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 (for 1KB BLKSIZE) block numbers, each 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 file systems.

The inode size (128 or 256) is designed to divides block size (1 KB or 4 KB) evenly, so that every inode block contains an integral number of inodes. In the simple EXT2 file system, the number of inodes is (a Linux default) 184. The number of inodes blocks is equal to 184/8 = 23. So the inodes blocks include B10 to B32. 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. Similarly, disk block numbers also count from 1 since block 0 is never used by a file system. A zero block number means no disk block.

11.2.7 Data Blocks

Data Blocks Immediately after the inodes blocks are data blocks for file storage. Assuming 184 inodes, the first real data block is B33, which is i_block[0] of the root directory /.

11.2.8 Directory Entries

EXT2 Directory Entries A directory contains dir_entry structures, which is

```
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 ending NULL
};
```

The dir_entry is an open-ended structure. The name field contains 1 to 255 chars without a terminating NULL. So the dir_entry's rec_len also varies.

11.3 Mailman's Algorithm

In computer systems, a problem which arises very often is as follows. A city has M blocks, numbered 0 to M-1. Each block has N houses, numbered 0 to N-1. Each house has a unique block address, denoted by (block, house), where 0 < = block < M, 0 < = house < N. An alien from outer space may be unfamiliar with the block addressing scheme on Earth and prefers to address the houses linearly as 0,1,..N-1,N, N + 1 ....., etc. Given a block address BA = (block, house), how to convert it to a linear address LA, and vice versa? If everything counts from 0, the conversion is very simple.

```
Linear_address LA = N*block + house;
Block_address  BA = (LA / N, LA % N);
```

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. The following shows applications of the Mailman's algorithm.

11.3.1 Test-Set-Clear Bits in C

(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.

11.3.2 Convert INODE Number to INODE on Disk

(2) 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.

11.4 Programming Examples

In this section, we shall show how to access and display the contents of EXT2 file systems by example programs. In order to compile and run these programs, the system must have the **ext2fs.h** header file installed, which defines the data structures of EXT2/3/4 file systems. Ubuntu Linux user may get and install the ext2fs development package by

```
sudo apt-get install ext2fs-dev
```

11.4.1 Display Superblock

The following C program displays the superblock of an EXT2 file system. The basic technique is as follows.

(1) Open vdisk for READ: int fd = open("vdisk", O_RDONLY);

(2) Read the superblock (block #1 or 1 KB at offset 1024) into a char buf[1024].

```
char buf[1024];
lseek(fd, 1024, SEEK_SET); // seek to byte offset 1024
int n = read(fd, buf, 1024);
```

(3) Let a struct ext2_super_block *sp point to buf[ ]. Then use sp- > field to access the various fields of the superblock structure.

```
struct ext2_super_block *sp = (struct ext2_super_block *)buf;
printf("s_magic = %x\n", sp->s_magic)                 // print s_magic
printf("s_inodes_count = %d\n", sp->s_inodes_count); // print s_inodes_
count
etc.
```

The same technique is also applicable to any other data structures in a real or virtual disk.

Example 11.1 superblock.c program: display superblock information of an EXT2 file system.

```
/*********** superblock.c program ************/
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <ext2fs/ext2_fs.h>

// typedef u8, u16, u32 SUPER for convenience
typedef unsigned char   u8;
typedef unsigned short u16;
typedef unsigned int   u32;
typedef struct ext2_super_block SUPER;

SUPER *sp;
char buf[1024];
int fd, blksize, inodesize;

int print(char *s, u32 x)
{
  printf("%-30s = %8d\n", s, x);
}
```

```
int super(char *device)
{
  fd = open(device, O_RDONLY);
  if (fd < 0){
    printf("open %sfailed\n", device); exit(1);
  }
  lseek(fd, (long)1024*1, 0);  // block 1 on FD, offset 1024 on HD
  read(fd, buf, 1024);
  sp = (SUPER *)buf;           // as a super block structure

  // check EXT2 FS magic number:
  printf("%-30s = %8x ", "s_magic", sp->s_magic);
  if (sp->s_magic != 0xEF53){
     printf("NOT an EXT2 FS\n");
     exit(2);
  }
  printf("EXT2 FS OK\n");
  print("s_inodes_count",      sp->s_inodes_count);
  print("s_blocks_count",      sp->s_blocks_count);
  print("s_r_blocks_count",   sp->s_r_blocks_count);
  print("s_free_inodes_count", sp->s_free_inodes_count);
  print("s_free_blocks_count", sp->s_free_blocks_count);
  print("s_first_data_block",  sp->s_first_data_block);
  print("s_log_block_size",    sp->s_log_block_size);
  print("s_blocks_per_group",  sp->s_blocks_per_group);
  print("s_inodes_per_group",  sp->s_inodes_per_group);
  print("s_mnt_count",         sp->s_mnt_count);
  print("s_max_mnt_count",     sp->s_max_mnt_count);
  printf("%-30s = %8x\n", "s_magic", sp->s_magic);
  printf("s_mtime = %s", ctime(&sp->s_mtime));
  printf("s_wtime = %s", ctime(&sp->s_wtime));
  blksize = 1024 * (1 << sp->s_log_block_size);
  printf("block size = %d\n", blksize);
  printf("inode size = %d\n",  sp->s_inode_size);
}

char *device = "mydisk";       // default device name
int main(int argc, char *argv[])
{
  if (argc>1)
     device = argv[1];
  super(device);
}
```

Figure 11.2 shows the outputs of running the superblock.c program.

```
s_magic                         =     ef53  EXT2 FS OK
s_inodes_count                  =      184
s_blocks_count                  =     1440
s__r_blocks_count               =       72
s_free_inodes_count             =      173
s_free_blocks_count             =     1393
s_first_data_blcok              =        1
s_log_block_size                =        0
s_blocks_per_group              =     8192
s_inodes_per_group              =      184
s_mnt_count                     =        1
s_max_mnt_count                 =       -1
s_magic                         =     ef53
s_mtime = Sun Oct  8 14:22:03 2017
s_wtime = Sun Oct  8 14:22:07 2017
block size = 1024
inode size = 128
```

Fig. 11.2 Superblock of Ext2 file system

Exercise 11.1 Write a C program to display the group descriptor of an EXT2 file system on a device. (See Problem 1).

11.4.2 Display Bitmaps

The program in Example 11.2 displays the inodes bitmap (imap) in HEX.

Example 11.2 imap.c program: display inodes bitmap of an EXT2 file system.

```
/*************** imap.c program **************/
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <ext2fs/ext2_fs.h>
typedef struct ext2_super_block SUPER;
typedef struct ext2_group_desc  GD;
#define BLKSIZE 1024
SUPER *sp;
GD    *gp;
char buf[BLKSIZE];
int fd;

// get_block() reads a disk block into a buf[ ]
int get_block(int fd, int blk, char *buf)
{
  lseek(fd, (long)blk*BLKSIZE, SEEK_SET);
  return read(fd, buf, BLKSIZE);
}

int imap(char *device)
{
  int i, ninodes, blksize, imapblk;
```

```
  fd = open(dev, O_RDONLY);
  if (fd < 0){printf("open %s failed\n", device); exit(1);}
  get_block(fd, 1, buf);          // get superblock
  sp = (SUPER *)buf;
  // check magic number to ensure it's an EXT2 FS
  ninodes = sp->s_inodes_count;   // get inodes_count
  printf("ninodes = %d\n", ninodes);
  get_block(fd, 2, buf);          // get group descriptor
  gp = (GD *)buf;
  imapblk = gp->bg_inode_bitmap;  // get imap block number
  printf("imapblk = %d\n", imapblk);
  get_block(fd, imapblk, buf);    // get imap block into buf[ ]
  for (i=0; i<=nidoes/8; i++){    // print each byte in HEX
      printf("%02x ", (u8)buf[i]);
  }
  printf("\n");
}

char * dev="mydisk";              // default device
int main(int argc, char *argv[ ] )
{
  if (argc>1) dev = argv[1];
  imap(dev);
}
```

The program prints each byte of the inodes bitmap as 2 HEX digits. The outputs look like the following.

```
|<-------------- niodes = 184 bits (23 bytes)---------------------|
ff 07 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ff
```

In the imap, bits are stored linearly from low to high address. The first 16 bits (from low to high) are b'11111111 11100000', but they are printed as ff 07 in HEX, which is not very informative since the bits in each byte are printed in reverse order, i.e. from high to low address.

Exercise 11.2 Modify the imap.c program to print the inodes bitmap in char map form, i.e. for each bit, print a '0' if the bit is 0, print a '1' if the bit is 1. (see Problem 2).

The outputs of the modified program should look like Fig. 11.3, in which each char represents a bit in the imap.

Exercise 11.3 Write a C program to display the blocks bitmap of an Ext2 file system, also in char map form.

```
ninodes = 184  imapblk = 9
11111111 11100000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000000
00000000 00000000 00000000 00000000 00000000 00000000 00000000 11111111
```

Fig. 11.3 Inodes Bitmap of an EXT2 file system

11.4.3 Display Root Inode

In an EXT2 file system, the number 2 (count from 1) inode is the inode of the root directory /. If we read the root inode into memory, we should be able to display its various fields such as mode, uid, gid, file size, time fields, hard links count and data block numbers, etc. The program in Example 11.3 displays the INDOE information of the root directory of an EXT2 file system.

Example 11.3 inode.c program: display root inode information of an EXT2 file system.

```
/*********** inode.c file **********/
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <ext2fs/ext2_fs.h>
#define BLKSIZE 1024
typedef struct ext2_group_desc  GD;
typedef struct ext2_super_block SUPER;
typedef struct ext2_inode       INODE;
typedef struct ext2_dir_entry_2 DIR;
SUPER *sp;
GD    *gp;
INODE *ip;
DIR   *dp;
char buf[BLKSIZE];
int fd, firstdata, inodesize, blksize, iblock;
char *dev = "mydisk";

int get_block(int fd, int blk, char *buf)
{
  lseek(fd, blk*BLKSIZE, SEEK_SET);
  return read(fd, buf, BLKSIZE);
}

int inode(char *dev)
{
  int i;
  fd = open(dev, O_RDONLY);
  if (fd < 0){
    printf("open failed\n"); exit(1);
  }
  /************************************
   same code as before to check EXT2 FS
  ************************************/
  get_block(fd, 2, buf);   // get group descriptor
  gp = (GD *)buf;
  printf("bmap_block=%d imap_block=%d inodes_table=%d ",
          gp->bg_block_bitmap,
          gp->bg_inode_bitmap,
          gp->bg_inode_table,
  iblock = gp->bg_inode_table;
```

```
  printf("---- root inode information ----\n");
  get_block(fd, iblock, buf);
  ip = (INODE *)buf;
  ip++;   // ip point at #2 INODE
  printf("mode = %4x ", ip->i_mode);
  printf("uid = %d  gid = %d\n", ip->i_uid, ip->i_gid);
  printf("size = %d\n", ip->i_size);
  printf("ctime = %s", ctime(&ip->i_ctime));
  printf("links = %d\n", ip->i_links_count);
  for (i=0; i<15; i++){         // print disk block numbers
    if (ip->i_block[i])         // print non-zero blocks only
       printf("i_block[%d] = %d\n", i, ip->i_block[i]);
  }
}

int main(int argc, char *argv[ ])
{
  if (argc>1) dev = argv[1];
  inode(dev);
}
```

Figure 11.4 shows the root inode of an EXT2 file system.

In Fig. 11.4, i_mode = 0x41ed or b′0100 0001 1110 1101′ in binary. The first 4 bits 0100 is the file type (DIRectory). The next 3 bits 000 = ugs are all 0's, meaning the file has no special usage, e.g. it is not a setuid program. The last 9 bits can be divided into 3 groups 111,101,101, which are the rwx permission bits of the file's owner, same group as owner and others. For regular files, x bit = 1 means the file is executable. For directories, x bit = 1 means access to (i.e. cd into) the directory is allowed; a 0 x bit means no access to the directory.

11.4.4 Display Directory Entries

Each data block of a directory INODE contains dir_entries, which are.

```
  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 ending NULL
  };
```

```
bmap = 8 imap = 9 iblock = 10
---- root inode information ----
mode = 41ed uid = 0  gid = 0
size = 1024
ctime = Mon Oct  9 16:11:44 2017
links = 3
i_block[0] = 33
```

Fig. 11.4 Root Inode of EXT2 file system

The contents of each data block of a directory has the form

[inode rec_len name_len NAME] [inode rec_len name_len NAME] ......

where NAME is a sequence of name_len chars without a terminating NULL byte. Each dir_entry has a record length rec_len. The rec_len of the last entry in a block covers the remaining block length, i.e. from where the entry begins to the end of block. The following algorithm shows how to step through the dir_entries in a directory data block.

```
/******** Algorithm to step through entries in a DIR data block ********/
struct ext2_dir_entry_2 *dp;             // dir_entry pointer
char *cp;                                // char pointer
int blk = a data block (number) of a DIR (e.g. i_block[0]);
char buf[BLKSIZE], temp[256];
get_block(fd, blk, buf);                 // get data block into buf[ ]

dp = (struct ext2_dir_entry_2 *)buf;     // as dir_entry
cp = buf;
while(cp < buf + BLKSIZE){
  strncpy(temp, dp->name, dp->name_len); // make name a string
  temp[dp->name_len] = 0;                // ensure NULL at end
  printf("%d %d %d %s\n", dp->inode, dp->rec_len, dp->name_len, temp);
  cp += dp->rec_len;                     // advance cp by rec_len
  dp = (struct ext2_dir_entry_2 *)cp;    // pull dp to next entry
}
```

Exercise 11.4 Write a C program to print the dir_entries of a directory. For simplicity, we may assume a directory INODE has at most 12 direct blocks, i_block[0] to i_block[11]. This assumption is reasonable. With 1 KB block size and an average file name length of 16 chars, a single disk block can contain up to 1024/(8 + 16) = 42 dir_entries. With 12 disk blocks, a directory can contain more than 500 entries. We may safely assume that no user would put that many files in any directory. For an empty EXT2 file system, the program output should look like Fig. 11.5. (see Problem 4).

```
check ext2 FS : OK
GD info: 8 9 10 1393 173 2
inodes begin block=10
******* root inode info ********
mode=41ed  uid=0  gid=0
size=1024
ctime=Sun Oct  8 14:04:52 2017
links=3
i_block[0]=33
********************************
inode# rec_len name_len name
  2       12       1    .
  2       12       2    ..
 11     1000      10 _  lost+found
```

Fig. 11.5 Entries of a directory

```
******* root inode info ********
mode=41ed  uid=0  gid=0
size=1024
ctime=Sun Oct  8 17:19:15 2017
links=7
i_block[0]=33
********************************
inode# rec_len name_len name
  2       12       1    .
  2       12       2    ..
 11       20      10    lost+found
 12       12       1    a
 13       16       8    shortDir
 14       20      12    longNamedDir
 15       28      17    aVeryLongNamedDir
 16       16       7    super.c
 17       16       6    bmap.c
 18       16       6    imap.c
 19      856       5    dir.c
```

Fig. 11.6 List entries of a directory

Exercise 11.5 Mount mydisk under Linux. Create new directories and copy files to the mounted file system, then umount it (See Problem 5). Run the dir.c program on mydisk again to see the outputs, which should look like Fig. 11.6. The reader may verify that the name_len of each entry is the exact number of chars in the name field, and every rec_len is a multiple of 4 (for alignment), which is (8 + name_len) raised to the next multiple of 4, except the last entry, whose rec_len covers the remaining block length.

Exercise 11.6 Given an INODE pointer to a DIRectory inode. Write a

```
int  search(INODE *dir, char *name)
```

function which searches for a dir_entry with a given name. Return its inode number if found, else return 0. (see Problem 6).

11.5 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.

11.5.1 Traversal Algorithm

(1) Read in the superblock. 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 12 (non-zero) direct blocks for name[0]. Each data block of a DIR INODE contains dir_entry structures of the form

```
[ino rec_len name_len NAME] [ino rec_len name_len NAME] ......
```

where NAME is a sequence of nlen chars without a terminating NULL. For each data block, read the block into memory and use a dir_entry *dp to point at the loaded data block. Then use name_len 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->rec_len);
```

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 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 current DIR 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 EXT2/3 file systems with many groups is similar. The reader may consult Chap. 3 of [Wang, 2015] for details.

11.5.2 Convert Pathname to INODE

Given a device containing an EXT2 file system and a pathname, .e.g. /a/b/c/d, write a C function.

```
INODE *path2inode(int fd, char *pathname)  // assume fd=file descriptor
```

which returns an INODE pointer to the file's inode or 0 if the file in inaccessible (see Problem 7).

As will be seen shortly, the path2inode() function is the most important function in a file system.

11.5.3 Display INODE Disk Blocks

Write a C program, showblock, which prints all disk block (numbers) of a file (see Problem 8).

11.6 Implementation of EXT2 File System

The objective of this section is to show how to implement a complete file system. First, we show the overall organization of a file system, and the logical steps of the implementation. Next, we present a base file system to help the reader get started. Then we organize the implementation steps as a series of programming projects for the reader to complete. It is believed that such an experience would be beneficial to any computer science student.

11.6.1 File System Organization

Figure 11.7 shows the internal organization of an EXT2 file system. The organization diagram is explained by the labels (1) to (5).

(1) is the PROC structure of the current running process. As in a real system, every file operation is due to the current executing process. Each PROC has a cwd, which points to the in-memory INODE of the PROC's Current Working Directory (CWD). It also has an array of file descriptors, fd[], which point to opened file instances.
(2) is the root pointer of the file system. It points to the in-memory root INODE. When the system starts, one of the devices is chosen as the root device, which must be a valid EXT2 file system. The root INODE (inode #2) of the root device is loaded into memory as the root (/) of the file system. This operation is known as **"mount root file system"**
(3) is an openTable entry. When a process opens a file, an entry of the PROC's fd array points to an openTable, which points to the in-memory INODE of the opened file.
(4) is an in-memory INODE. Whenever a file is needed, its INODE is loaded into a minode slot for reference. Since INODEs are unique, only one copy of each INODE can be in memory at any time. In the minode, (dev, ino) identify where the INODE came from, for writing the INODE back to disk if modified. The refCount field records the number of processes that are using the minode.

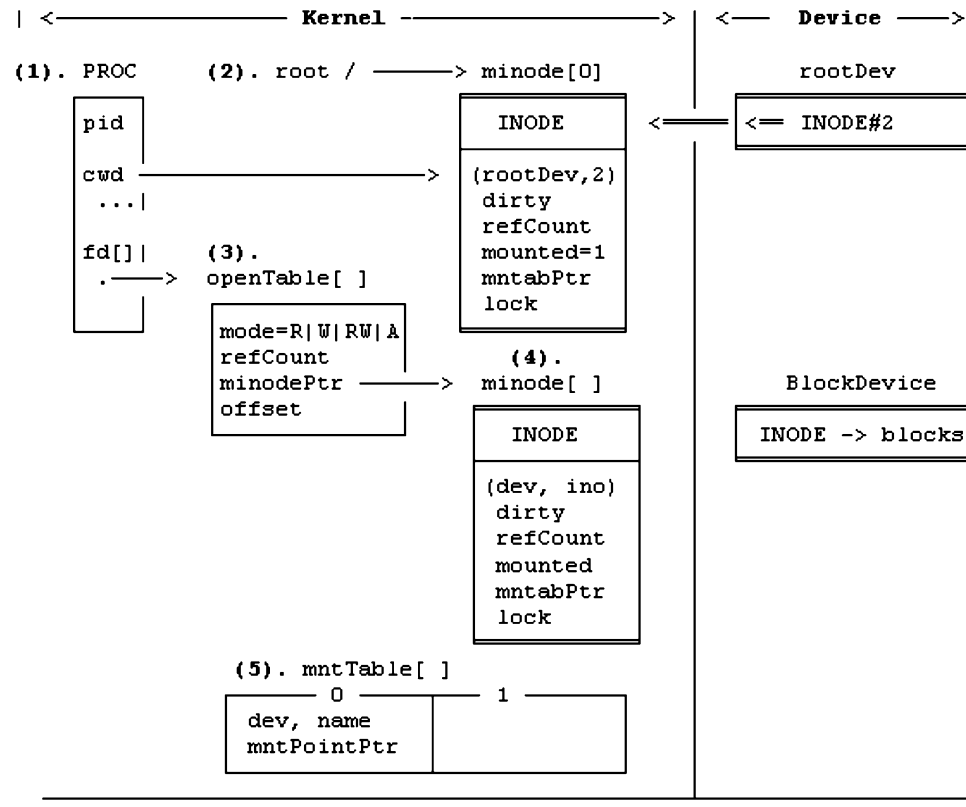


Fig. 11.7 EXT2 file system data structures

The dirty field indicates whether the INODE has been modified. The mounted flag indicates whether the INODE has been mounted on and, if so, the mntabPtr points to the mount table entry of the mounted file system. The lock field is to ensure that an in-memory INODE can only be accessed by one process at a time, e.g. when modifying the INODE or during a read/write operation.

(5) is a table of mounted file systems. For each mounted file system, an entry in the mount table is used to record the mounted file system information, e.g. the mounted file system device number. In the in-memory INODE of the mount point, the mounted flag is turned on and the mntabPtr points to the mount table entry. In the mount table entry, mntPointPtr points back to the in-memory INODE of the mount point. As will be shown later, these doubly-linked pointers allow us to cross mount points when traversing the file system tree. In addition, a mount table entry may also contain other information of the mounted file system, such as values from the superblock, group descriptor, bitmaps and inodes start block, etc. for quick access. If any of these cached items are modified, they must be written back to the device when the device is umounted.

11.6.2 Files System Levels

Implementation of the file system is divided into three levels. Each level deals with a distinct part of the file system. This makes the implementation process modular and easier to understand. In the file system implementation, the FS directory contains files which implement the EXT2 file system. The files are organized as follows.

```
------------------ Common files of FS -----------------------------
  type.h  : EXT2 data structure types
  global.c: global variables of FS
  util.c    : common utility functions: getino(), iget(), iput(), search
  (), etc.
  allocate_deallocate.c : inodes/blocks management functions
```

Level-1 implements the basic file system tree. It contains the following files, which implement the indicated functions.

```
------------------- Level-1 of FS ----------------------------
mkdir_creat.c       : make directory, create regular file
ls_cd_pwd.c         : list directory, change directory, get CWD path
rmdir.c             : remove directory
link_unlink.c       : hard link and unlink files
symlink_readlink.c  : symbolic link files
stat.c              : return file information
misc1.c             : access, chmod, chown, utime, etc.
-------------------------------------------------------------------
```

User command programs which use the level-1 FS functions include

mkdir, creat, mknod, rmdir, link, unlink, symlink, rm, ls, cd and pwd, etc.

Level-2 implements functions for reading/writing file contents.

```
---------------------- Level-2 of FS ------------------------------
open_close_lseek.c   : open file for READ|WRITE|APPEND, close file and lseek
read.c               : read from file descriptor of an opened regular file
write.c              : write to file descriptor of an opened regular file
opendir_readdir.c    : open and read directory
```

Level-3 implements mount, umount file systems and file protection.

```
---------------------- Level-3 of FS ------------------------------
mount_umount.c      : mount/umount file systems
file protection     : access permission checking
file-locking        : lock/unlock files
-------------------------------------------------------------------
```

11.7 Base File System

11.7.1 type.h file

This file contains the data structure types of the EXT2 file system, such as superblock, group descriptor, inode and directory entry structures. In addition, it also contains the open file table, mount table, PROC structures and constants of the file system.

```
/***** type.h file for EXT2 FS *****/
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <linux/ext2_fs.h>
#include <libgen.h>
#include <string.h>
#include <sys/stat.h>
// define shorter TYPES for convenience
typedef struct ext2_group_desc  GD;
typedef struct ext2_super_block SUPER;
typedef struct ext2_inode       INODE;
typedef struct ext2_dir_entry_2 DIR;
#define BLKSIZE        1024

// Block number of EXT2 FS on FD
#define SUPERBLOCK        1
#define GDBLOCK           2
#define ROOT_INODE        2

// Default dir and regular file modes
#define DIR_MODE          0x41ED
#define FILE_MODE         0x81AE
#define SUPER_MAGIC       0xEF53
#define SUPER_USER        0
// Proc status
#define FREE              0
#define BUSY              1

// file system table sizes
#define NMINODE          100
#define NMTABLE           10
#define NPROC              2
#define NFD               10
#define NOFT              40

// Open File Table
typedef struct oft{
  int   mode;
  int   refCount;
  struct minode *minodePtr;
  int   offset;
}OFT;

// PROC structure
typedef struct proc{
  struct Proc *next;
  int   pid;
  int   uid;
  int   gid;
```

```
  int   ppid;
  int   status;
  struct minode *cwd;
  OFT   *fd[NFD];
}PROC;

// In-memory inodes structure
typedef struct minode{
  INODE INODE;            // disk inode
  int   dev, ino;
  int   refCount;         // use count
  int   dirty;            // modified flag
  int   mounted;          // mounted flag
  struct mount *mntPtr;   // mount table pointer
  // int lock;            // ignored for simple FS
}MINODE;

// Open file Table        // opened file instance
typedef struct oft{
  int  mode;              // mode of opened file
  int  refCount;          // number of PROCs sharing this instance
  MINODE *minodePtr;      // pointer to minode of file
  int  offset;            // byte offset for R|W
}OFT;

// Mount Table structure
typedef struct mtable{
  int    dev;            // device number; 0 for FREE
  int    ninodes;        // from superblock
  int    nblocks;
  int    free_blocks     // from superblock and GD
  int    free_inodes
  int    bmap;           // from group descriptor
  int    imap;
  int    iblock;         // inodes start block
  MINODE *mntDirPtr;     // mount point DIR pointer
  char   devName[64];    //device name
  char   mntName[64];    // mount point DIR name
}MTABLE;
```

(2). global.c file: This file contains global variables of the file system. Examples of global variables are

```
  MINODE minode[NMINODE];     // in memory INODEs
  MTABLE mtable[NMTABLE];     // mount tables
  OFT    oft[NOFT];           // Opened file instance
  PROC   proc[NPROC]          // PROC structures
  PROC   *running;            // current executing PROC
```

When the file system starts, we initializes all the global data structures and let running point at PROC[0], which is process P0 of the superuser (uid = 0). As in a real file system, every operation is due to the current running process. We begin with a superuser process because it does not need any file protection. File protection by permission checking will be enforced later in Level 3 of the FS implementation.

```
int fs_init()
{
  int i,j;
  for (i=0; i<NMINODE; i++)     // initialize all minodes as FREE
      minode[i].refCount = 0;
  for (i=0; i<NMTABLE; i++)     // initialize mtables as FREE
      mtable[i].dev = 0;
  for (i=0; i<NOFT; i++)        // initialize ofts as FREE
      oft[i].refCount = 0;
  for (i=0; i<NPROC; i++){      // initialize PROCs
      proc[i].status = READY;   // ready to run
      proc[i].pid = i;          // pid = 0 to NPROC-1
      proc[i].uid = i;          // P0 is a superuser process
      for (j=0; j<NFD; j++)
          proc[i].fd[j] = 0;    // all file descriptors are NULL
      proc[i].next = &proc[i+1];   // link list
  }
  proc[NPROC-1].next = &proc[0];   // circular list
  running = &proc[0];              // P0 runs first
}
```

During file system operation, global data structures are regarded as system resources, which are used and released dynamically. Each set of resources is managed by a pair of allocate and deallocate functions. For example, mialloc() allocates a FREE minode for use, and midalloc() releases a used minode. Other resource management functions are similar, which will be shown later when they are actually needed.

```
MINODE *mialloc()          // allocate a FREE minode for use
{
   int i;
   for (i=0; i<NMINODE; i++){
     MIONDE *mp = &minode[i];
     if (mp->refCount == 0){
        mp->refCount = 1;
        return mp;
     }
   }
   printf("FS panic: out of minodes\n");
   return 0;
}
int midalloc(MINODE *mip) // release a used minode
{
  mip->refCount = 0;
}
```

11.7.2 Utility Functions

(3). util.c file: This file contains commonly used utility functions of the file system. The most important utility functions are read/write disk block functions, iget(), iput() and getino(), which are explained in more detail.

(3).1. get_block/put_block functions: We assume that a block device, e.g. a real or virtual disk, can only be read or written in unit of block size. For real disks, this is due to hardware constraints. For virtual disks, we assume that read/write is also by block size, so that the code can be ported to real disks if desired. For a virtual disk, we first open it for R|W mode and use the file descriptor as the device number. The following functions read/write a virtual disk block into/from a buffer area in memory.

```
int get_block(int dev, int blk, char *buf)
{
    lseek(dev, blk*BLKSIZE, SEEK_SET);
    int n = read(dev, buf, BLKSIZE);
    if (n<0) printf("get_block [%d %d] error\n", dev, blk);
}
int put_block(int dev, int blk, char *buf)
{
    lseek(dev, blk*BLKSIZE, SEEK_SET);
    int n = write(dev, buf, BLKSIZE);
    if (n != BLKSIZE)
       printf("put_block [%d %d] error\n", dev, blk);
}
```

(3).2. iget(dev, ino) function: This function returns a pointer to the in-memory minode containing the INODE of (dev, ino). The returned minode is unique, i.e. only one copy of the INODE exists in memory. In a real file system, the returned minode is locked for exclusive use until it is either released or unlocked. For simplicity, we shall assume that minode locking is unnecessary, which will be explained later.

```
MINODE *iget(int dev, int ino)
{
  MINODE *mip;
  MTABLE *mp;
  INODE  *ip;
  int i, block, offset;
  char buf[BLKSIZE];

  // serach in-memory minodes first
  for (i=0; i<NMINODES; i++){
    MINODE *mip = &MINODE[i];
    if (mip->refCount && (mip->dev==dev) && (mip->ino==ino)){
       mip->refCount++;
       return mip;
    }
  }
```

```
  // needed INODE=(dev,ino) not in memory
  mip = mialloc();                  // allocate a FREE minode
  mip->dev = dev; mip->ino = ino;   // assign to (dev, ino)
  block = (ino-1)/8 + iblock;       // disk block containing this inode
  offset= (ino-1)%8;                // which inode in this block
  get_block(dev, block, buf);
  ip = (INODE *)buf + offset;
  mip->INODE = *ip;                 // copy inode to minode.INODE
  // initialize minode
  mip->refCount = 1;
  mip->mounted = 0;
  mip->dirty = 0;
  mip->mountptr = 0;
  return mip;
}
```

(3).3. The iput(INODE *mip) function: This function releases a used minode pointed by mip. Each minode has a refCount, which represents the number of users that are using the minode. iput() decrements the refCount by 1. If the refCount is non-zero, meaning that the minode still has other users, the caller simply returns. If the caller is the last user of the minode (refCount = 0), the INODE is written back to disk if it is modified (dirty).

```
int iput(MINODE *mip)
{
  INODE  *ip;
  int i, block, offset;
  char buf[BLKSIZE];

  if (mip==0) return;
  mip->refCount--;                  // dec refCount by 1
  if (mip->refCount > 0) return;    // still has user
  if (mip->dirty == 0)   return;    // no need to write back

  // write INODE back to disk
  block  = (mip->ino - 1) / 8 + iblock;
  offset = (mip->ino - 1) % 8;

  // get block containing this inode
  get_block(mip->dev, block, buf);
  ip = (INODE *)buf + offset;       // ip points at INODE
  *ip = mip->INODE;                 // copy INODE to inode in block
  put_block(mip->dev, block, buf); // write back to disk
  midalloc(mip);                    // mip->refCount = 0;
}
```

(3).4. getino() function: The getino() function implements the file system tree traversal algorithm. It returns the INODE number (ino) of a specified pathname. To begin with, we assume that in the level-1 file system implementation, the file system resides on a single root device, so that there are no mounted devices and mounting point crossings. Mounted file systems and mounting point crossing will be

considered later in level-3 of the file system implementation. Thus, the getino() function essentially returns the (dev, ino) of a pathname. The function first uses the tokenize() function to break up pathname into component strings. We assume that the tokenized strings are in a global data area, each pointed by a name[i] pointer and the number of token strings is nname. Then it calls the search() function to search for the token strings in successive directories. The following shows the tokenize() and search() functions.

```
char *name[64];  // token string pointers
char gline[256]; // holds token strings, each pointed by a name[i]
int  nname;      // number of token strings

int tokenize(char *pathname)
{
  char *s;
  strcpy(gline, pathname);
  nname = 0;
  s = strtok(gline, "/");
  while(s){
    name[nname++] = s;
    s = strtok(0, "/");
  }
}

int search(MINODE *mip, char *name)
{
   int i;
   char *cp, temp[256], sbuf[BLKSIZE];
   DIR *dp;
   for (i=0; i<12; i++){ // search DIR direct blocks only
     if (mip->INODE.i_block[i] == 0)
        return 0;
     get_block(mip->dev, mip->INODE.i_block[i], sbuf);
     dp = (DIR *)sbuf;
     cp = sbuf;
     while (cp < sbuf + BLKSIZE){
        strncpy(temp, dp->name, dp->name_len);
        temp[dp->name_len] = 0;
        printf("%8d%8d%8u %s\n",
               dp->inode, dp->rec_len, dp->name_len, temp);
        if (strcmp(name, temp)==0){
           printf("found %s : inumber = %d\n", name, dp->inode);
           return dp->inode;
        }
        cp += dp->rec_len;
        dp = (DIR *)cp;
     }
   }
   return 0;
}
```

```
int getino(char *pathname)
{
  MINODE *mip;
  int i, ino;
  if (strcmp(pathname, "/")==0){
      return 2;              // return root ino=2
  }
  if (pathname[0] == '/')
      mip = root;            // if absolute pathname: start from root
  else
      mip = running->cwd;    // if relative pathname: start from CWD
  mip->refCount++;           // in order to iput(mip) later

  tokenize(pathname);        // assume: name[ ], nname are globals

  for (i=0; i<nname; i++){   // search for each component string
      if (!S_ISDIR(mip->INODE.i_mode)){  // check DIR type
         printf("%s is not a directory\n", name[i]);
         iput(mip);
         return 0;
      }
      ino = search(mip, name[i]);
      if (!ino){
         printf("no such component name %s\n", name[i]);
         iput(mip);
         return 0;
      }
      iput(mip);               // release current minode
      mip = iget(dev, ino);    // switch to new minode
  }
  iput(mip);
  return ino;
}
```

(3).5. Use of getino()/iget()/iput(): In a file system, almost every operation begins with a pathname, e.g. mkdir pathname, cat pathname, etc. Whenever a pathname is specified, its inode must be loaded into memory for reference. The general pattern of using an inode is

```
. ino  = getino(pathname);
. mip = iget(dev, ino);
.    use mip->INODE, which may modify the INODE;
. iput(mip);
```

There are only a few exceptions to this usage pattern. For instance,

chdir : iget the new DIR minode but iput the old DIR minode.
open : iget the minode of a file, which is released when the file is closed.
mount: iget the minode of mountPoint, which is released later by umount.

In general, iget and iput should appear in pairs, like a pair of matched parentheses. We may rely on this usage pattern in the implementation code to ensure that every INODE is loaded and then released properly.

(4). Minodes Locking: In a real file system, each minode has a lock field, which ensures that the minode can only be accessed by one process at a time, e.g. when modifying the INODE. Unix kernel uses a busy flag and sleep/wakeup to synchronize processes accessing the same minode. In other systems, each minode may have a mutex lock or a semaphore lock. A process is allowed to access a minode only if it holds the minode lock. The reason for minodes locking is as follows.

Assume that a process Pi needs the inode of (dev, ino), which is not in memory. Pi must load the inode into a minode entry. The minode must be marked as (dev, ino) to prevent other processes from loading the same inode again. While loading the inode from disk Pi may wait for I/O completion, which switches to another process Pj. If Pj needs exactly the same inode, it would find the needed minode already exists. Without the lock, Pj would proceed to use the minode before it is even loaded in yet. With the lock, Pj must wait until the minode is loaded, used and then released by Pi. In addition, when a process read/write an opened file, it must lock the file's minode to ensure that each read/write operation is atomic. For simplicity, we shall assume that only one process runs at a time, so that the lock is unnecessary. However, the reader should beware that minodes locking is necessary in a real file system.

Exercise 11.7 Design and implement a scheme, which uses the minode[ ] area as a cache memory for in-memory INODES. Once an INODE is loaded into a minode slot, try to keep it in memory for as long as possible even if no process is actively using it.

11.7.3 Mount-Root

(5). mount_root.c file: This file contains the mount_root() function, which is called during system initialization to mount the root file system. It reads the superblock of the root device to verify the device is a valid EXT2 file system.Then it loads the root INODE (ino = 2) of the root device into a minode and sets the root pointer to the root minode. It also sets the CWD of all PROCs to the root minode. A mount table entry is allocated to record the mounted root file system. Some key information of the root device, such as the number of inodes and blocks, the starting blocks of the bitmaps and inodes table, are also recorded in the mount table for quick access.

```
/************ FS1.1.c file ************/
#include "type.h"
#include "util.c"

// global variables
MINODE minode[NMINODES], *root;
MTABLE mtable[NMTABLE];
PROC proc[NPROC], *running;
int ninode, nblocks, bmap, imap, iblock;
int dev;
char gline[25], *name[16];  // tokenized component string strings
int  nname;                 // number of component strings
char *rootdev = "mydisk";   // default root_device
```

```
int fs_init()
{
  int i,j;
  for (i=0; i<NMINODES; i++)  // initialize all minodes as FREE
      minode[i].refCount = 0;
  for (i=0; i<NMOUNT; i++)    // initialize mtable entries as FREE
      mtable[i].dev = 0;
  for (i=0; i<NPROC; i++){    // initialize PROCs
      proc[i].status = READY; // reday to run
      proc[i].pid = i;        // pid = 0 to NPROC-1
      proc[i].uid = i;        // P0 is a superuser process
      for (j=0; j<NFD; j++)
          proc[i].fd[j] = 0;  // all file descriptors are NULL
      proc[i].next = &proc[i+1];
  }
  proc[NPROC-1].next = &proc[0];   // circular list
  running = &proc[0];              // P0 runs first
}

int mount_root(char *rootdev)  // mount root file system
{
  int i;
  MTABLE *mp;
  SUPER  *sp;
  GD     *gp;
  char buf[BLKSIZE];

  dev = open(rootdev, O_RDWR);
  if (dev < 0){
     printf("panic : can't open root device\n");
     exit(1);
  }
  /* get super block of rootdev */
  get_block(dev, 1, buf);
  sp = (SUPER *)buf;
  /* check magic number */
  if (sp->s_magic != SUPER_MAGIC){
     printf("super magic=%x : %s is not an EXT2 filesys\n",
             sp->s_magic, rootdev);
     exit(0);
  }
  // fill mount table mtable[0] with rootdev information
  mp = &mtable[0];     // use mtable[0]
  mp->dev = dev;
  // copy super block info into mtable[0]
  ninodes = mp->ninodes = sp->s_inodes_count;
  nblocks = mp->nblocks = sp->s_blocks_count;
  strcpy(mp->devName, rootdev);
  strcpy(mp->mntName, "/");
```

```
  get_block(dev, 2, buf);
  gp = (GD *)buf;
  bmap = mp->bmap = gp->bg_blocks_bitmap;
  imap = mp->imap = gp->bg_inodes_bitmap;
  iblock = mp->iblock = gp->bg_inode_table;
  printf("bmap=%d imap=%d iblock=%d\n", bmap, imap iblock);

  // call iget(), which inc minode's refCount
  root = iget(dev, 2);           // get root inode
  mp->mntDirPtr = root;          // double link
  root->mntPtr = mp;
  // set proc CWDs
  for (i=0; i<NPROC; i++)         // set proc's CWD
      proc[i].cwd = iget(dev, 2); // each inc refCount by 1
  printf("mount : %s  mounted on / \n", rootdev);
  return 0;
}

int main(int argc, char *argv[ ])
{
  char line[128], cmd[16], pathname[64];
  if (argc > 1)
     rootdev = argv[1];
  fs_init();
  mount_root(rootdev);
  while(1){
     printf("P%d running: ", running->pid);
     printf("input command : ");
     fgets(line, 128, stdin);
     line[strlen(line)-1] = 0;
     if (line[0]==0)
        continue;
     sscanf(line, "%s %s", cmd, pathname);
     if (!strcmp(cmd, "ls"))
        ls(pathname);
     if (!strcmp(cmd, "cd"))
        chdir(pathname);
     if (!strcmp(cmd, "pwd"))
        pwd(running->cwd);
     if (!strcmp(cmd, "quit"))
        quit();
  }
}

int quit() // write all modified minodes to disk
{
   int i;
   for (i=0; i<NMINODES; i++){
     MINODE *mip = &minode[i];
     if (mip->refCount && mip->dirty){
```

```
        mip->refCount = 1;
        iput(mip);
      }
   }
   exit(0);
}
```

How to ls: ls [pathname] lists the information of either a directory or a file. In Chap. 5 (Sect. 5.4), we showed an ls program, which works as follows.

(1) ls_dir(dirname): use opendir() and readdir() to get filenames in the directory. For each filename, call ls_file(filename).

(2) ls_file(filename): stat the filename to get file information in a STAT structure. Then list the STAT information.

Since the stat system call essentially returns the same information of a minode, we can modify the original ls algorithm by using minodes directly. The following shows the modified ls algorithm

```
/************************ Algorithm of ls ******************************/
(1). From the minode of a directory, step through the dir_entries in the data
blocks of the minode.INODE. Each dir_entry contains the inode number, ino, and
name of a file. For each dir_entry, use iget() to get its minode, as in
         MINODE *mip = iget(dev, ino);
Then, call ls_file(mip, name).
(2). ls_file(MINODE *mip, char *name): use mip->INODE and name to list the file
information.
```

How to chdir [pathname]: The algorithm of chdir is as follows.

```
/*************** Algorithm of chdir ****************/
(1). int ino = getino(pathname);          // return error if ino=0
(2). MINODE *mip = iget(dev, ino);
(3). Verify mip->INODE is a DIR      // return error if not DIR
(4). iput(running->cwd);                    // release old cwd
(5). running->cwd = mip;                   // change cwd to mip
```

HOW TO pwd: The following shows the algorithm of pwd, which uses recursion on the directory minodes.

```
/*************** Algorithm of pwd ****************/
   rpwd(MINODE *wd){
    (1). if (wd==root) return;
    (2). from wd->INODE.i_block[0], get my_ino and parent_ino
    (3). pip = iget(dev, parent_ino);
    (4). from pip->INODE.i_block[ ]: get my_name string by my_ino as LOCAL
    (5). rpwd(pip);  // recursive call rpwd(pip) with parent minode
    (6). print "/%s", my_name;
   }
```

```
pwd(MINODE *wd){
   if (wd == root)   print "/"
   else                    rpwd(wd);
}
// pwd start:
pwd(running->cwd);
```

Exercise 11.8 Implement step (2) of the pwd algorithm as a utility function.

```
int get_myino(MINODE *mip, int *parent_ino)
```

which returns the inode number of . and that of .. in parent_ino.

Exercise 11.9 Implement step (4) of the pwd algorithm as a utility function.

```
int get_myname(MINODE *parent_minode, int my_ino, char *my_name)
```

which returns the name string of a dir_entry identified by my_ino in the parent directory.

11.7.4 Implementation of Base File System

The programming task here is to complete the above mount_root.c program as the base file system. Run it with a virtual disk containing an EXT2 file system. The outputs of the base file system should look like Fig. 11.8. The figure only shows the results of the ls command. It should also support the cd and pwd commands.

```
checking EXT2 FS ....OK
bmp = 8 imap = 9 inodes_start = 10
init()
mount_root()
creating P0 as running process
mydisk mounted on / OK
input command : [ls|cd|pwd|quit] ls
drwxr-xr-x   5   0   0  Oct 21 09:02    1024    .
drwxr-xr-x   5   0   0  Oct 21 09:02    1024    ..
drwx------   2   0   0  Oct 21 09:01   12288    lost+found
drwxr-xr-x   4   0   0  Oct 21 09:03    1024    dir1
drwxr-xr-x   2   0   0  Oct 21 09:02    1024    dir2
-rw-r--r--   1   0   0  Oct 21 09:02       0    file1
-rw-r--r--   1   0   0  Oct 21 09:02       0    file2
input command : [ls|cd|pwd|quit]
```

Fig. 11.8 Sample outputs of mount_root

11.8 File System Level-1 Functions

1. mkdir: The command

```
mkdir pathname
```

makes a new directory with pathname. The permission bits of the new directory are set to the default value 0755 (owner can access and rw, others can access but can only READ).

11.8.1 Algorithm of mkdir

mkdir creates an empty directory with a data block containing the default . and .. entries. The algorithm of mkdir is

```
/********* Algorithm of mkdir pathname*********/
(1). divide pathname into dirname and basename, e.g. pathname=/a/b/c, then
          dirname=/a/b;   basename=c;
(2). // dirname must exist and is a DIR:
          pino = getino(dirname);
          pmip = iget(dev, pino);
          check pmip->INODE is a DIR
(3). // basename must not exist in parent DIR:
         search(pmip, basename) must return 0;
(4). call kmkdir(pmip, basename) to create a DIR;

       kmkdir() consists of 4 major steps:
       (4).1. Allocate an INODE and a disk block:
                ino = ialloc(dev);
                blk = balloc(dev);
       (4).2. mip = iget(dev, ino) // load INODE into a minode
                initialize mip->INODE as a DIR INODE;
                mip->INODE.i_block[0] = blk; other i_block[ ] = 0;
                mark minode modified (dirty);
                iput(mip);  // write INODE back to disk
       (4).3. make data block 0 of INODE to contain . and .. entries;
              write to disk block blk.
       (4).4. enter_child(pmip, ino, basename); which enters
                (ino, basename) as a dir_entry to the parent INODE;

(5). increment parent INODE's links_count by 1 and mark pmip dirty;
     iput(pmip);
```

Most steps of the mkdir algorithm are self-explanatory. Only step (4) needs more explanation. We illustrate step (4) in more detail by example code. In order to make a directory, we need to allocate an inode from the inodes bitmap, and a disk block from the blocks bitmap, which rely on test and set bits in the bitmaps. In order to maintain file system consistency, allocating an inode must decrement the free inodes count in both superblock and group descriptor by 1. Similarly, allocating a disk block must decrement the free blocks count in both superblock and group descriptor by 1. It is also worth noting

that bits in the bitmaps count from 0 but inode and block numbers count from 1. The following code segments show the detailed steps of (4).

(4).1 Allocate Inode and Disk Block:

```
// tst_bit, set_bit functions
int tst_bit(char *buf, int bit){
    return buf[bit/8] & (1 << (bit % 8));
}
int set_bit(char *buf, int bit){
    buf[bit/8] |= (1 << (bit % 8));
}
int decFreeInodes(int dev)
{
  // dec free inodes count in SUPER and GD
  get_block(dev, 1, buf);
  sp = (SUPER *)buf;
  sp->s_free_inodes_count--;
  put_block(dev, 1, buf);
  get_block(dev, 2, buf);
  gp = (GD *)buf;
  gp->bg_free_inodes_count--;
  put_block(dev, 2, buf);
}
int ialloc(int dev)
{
 int i;
 char buf[BLKSIZE];
 // use imap, ninodes in mount table of dev
 MTABLE *mp = (MTABLE *)get_mtable(dev);
 get_block(dev, mp->imap, buf);
 for (i=0; i<mp->ninodes; i++){
   if (tst_bit(buf, i)==0){
      set_bit(buf, i);
      put_block(dev, mp->imap, buf);
      // update free inode count in SUPER and GD
      decFreeInodes(dev);
      return (i+1);
   }
 }
 return 0; // out of FREE inodes
}
```

Allocation of disk blocks is similar, except it uses the blocks bitmap and it decrements the free blocks count in both superblock and group descriptor. This is left as an exercise.

Exercise 11.10 Implement the function

```
int balloc(int dev)
```

which allocates a free disk block (number) from a device.

(4).2. Create INODE: The following code segment creates an INODE=(dev, ino) in a minode, and writes the INODE to disk.

```
MINODE *mip = iget(dev, ino);
INODE *ip = &mip->INODE;
ip->i_mode = 0x41ED;       // 040755: DIR type and permissions
ip->i_uid  = running->uid; // owner uid
ip->i_gid  = running->gid; // group Id
ip->i_size = BLKSIZE;      // size in bytes
ip->i_links_count = 2;     // links count=2 because of . and ..
ip->i_atime = ip->i_ctime = ip->i_mtime = time(0L);
ip->i_blocks = 2;          // LINUX: Blocks count in 512-byte chunks
ip->i_block[0] = bno;      // new DIR has one data block
ip->i_block[1] to ip->i_block[14] = 0;
mip->dirty = 1;            // mark minode dirty
iput(mip);                 // write INODE to disk
```

(4).3. Create data block for new DIR containing . and .. entries

```
char buf[BLKSIZE];
bzero(buf, BLKSIZE); // optional: clear buf[ ] to 0
DIR *dp = (DIR *)buf;
// make . entry
dp->inode = ino;
dp->rec_len = 12;
dp->name_len = 1;
dp->name[0] = '.';
// make .. entry: pino=parent DIR ino, blk=allocated block
dp = (char *)dp + 12;
dp->inode = pino;
dp->rec_len = BLKSIZE-12;   // rec_len spans block
dp->name_len = 2;
dp->name[0] = d->name[1] = '.';
put_block(dev, blk, buf);   // write to blk on diks
```

(4).4. Enter new dir_entry into parent directory: The function

```
int enter_name(MINODE *pip,  int ino,  char *name)
```

enters a [ino, name] as a new dir_entry into a parent directory. The enter_name algorithm consists of several steps.

```
/****************** Algorithm of enter_name ******************/
 for each data block of parent DIR do  // assume: only 12 direct blocks
{
        if (i_block[i]==0) BREAK;     // to step (5) below
   (1). Get parent's data block into a buf[ ];
   (2). In a data block of the parent directory, each dir_entry has an ideal
   length
```

```
        ideal_length = 4*[ (8 + name_len + 3)/4 ]     // a multiple of 4

        All dir_entries rec_len = ideal_length, except the last entry. The
        rec_len of the LAST entry is to the end of the block, which may be
        larger than its ideal_length.
    (3). In order to enter a new entry of name with n_len, the needed length is

        need_length = 4*[ (8 + n_len + 3)/4 ]         // a multiple of 4

    (4). Step to the last entry in the data block:

        get_block(parent->dev, parent->INODE.i_block[i], buf);
        dp = (DIR *)buf;
        cp = buf;
        while (cp + dp->rec_len < buf + BLKSIZE){
           cp += dp->rec_len;
           dp = (DIR *)cp;
        }
        // dp NOW points at last entry in block
        remain = LAST entry's rec_len - its ideal_length;

        if (remain >= need_length){
           enter the new entry as the LAST entry and
           trim the previous entry rec_len to its ideal_length;
        }
        goto step (6);
      }
```

The following diagrams show the block contents before and after entering [ino, name] as a new entry.

```
Before                  |         LAST entry                    |
|-4---2----2--|----|----|------- rlen to end of block ----------|
|ino rlen nlen NAME|.....|ino rlen nlen|NAME                     |
-------------------------------------------------------------------

After                   |    Last entry    |     NEW entry
|-4---2----2--|----|----|----ideal_len-----|---- rlen=remain ----|
|ino rlen nlen NAME|....|ino rlen nlen|NAME|ino rlen nlen name   |
 -------------------------------------------------------------------
}
```

(5). If no space in existing data block(s):

Allocate a new data block; increment parent size by BLKSIZE;
Enter new entry as the first entry in the new data block with rec_len=BLKSIZE.

The following diagrams shows the new data block containing only one entry.

```
 |------------------- rlen = BLKSIZE -------------------------------
 |ino rlen=BLKSIZE nlen name                                        |
 ---------------------------------------------------------------------
 (6).Write data block to disk;
}
```

2. creat : Creat creates an empty regular file. The algorithm of creat is

11.8.2 Algorithm of creat

```
/************** Algorithm of creat  *************/
creat(char * pathname)
{
  This is similar to mkdir() except
  (1). the INODE.i_mode field is set to REG file type, permission bits set to
               0644 = rw-r--r--, and
  (2). no data block is allocated for it, so the file size is 0.
  (3). links_count = 1; Do not increment parent INODE's links_count
}
```

It is noted that the above creat algorithm differs from Unix/Linux in that it does not open the file for WRITE mode and return a file descriptor. In practice, creat is rarely used as a stand-alone function. It is used internally by the open() function, which may create a file, open it for WRITE and return a file descriptor. The open operation will be implemented later in level-2 of the file system.

11.8.3 Implementation of mkdir-creat

The programming task here is to implement **mkdir-creat** functions and add the corresponding mkdir and creat commands to the base file system. Demonstrate the system by testing the mkdir and creat functions. Figure 11.9 shows the sample outputs of the resulting file system. It shows the detailed steps of executing the command **mkdir dir1,** as described in the mkdir Algorithm.

3. rmdir: The command

```
rmdir dirname
```

removes a directory. As in Unix/Linux, in order to remove a DIR, the directory must be empty, for the following reasons. First, removing a non-empty directory implies the removal of all the files and subdirectories in the directory. Although it is possible to implement a recursive rmdir operation, which removes an entire directory tree, the basic operation is still remove one directory at a time. Second, a non-empty directory may contain files that are actively in use, e.g. files opened for read/write, etc. Removing such a directory is clearly unacceptable. Although it is possible to check whether there are any active files in a directory, doing this would incur too much overhead. The simplest way out is to require that a directory must be empty in order to be removed. The algorithm of rmdir is.

```
fs_init()
mount_root
checking EXT2 FS : OK
bmp = 8 imap = 9 iblock = 10
mydisk mounted on / OK
root refCount = 3
input command : [ls|cd|pwd|mkdir|creat|quit] mkdir dir1
mkdir dir1
parent=.  child=dir1
getino: pathname=.
========================================
getino: i=0 name[0]=.
search for . in MINODE = [3, 2]
. found . : ino = 2
search for dir1 in MINODE = [3, 2]
. .. lost+found : dir1 not yet exits
ialloc: ino=12 balloc: bno=47
making INODE
iput: dev=3 ino=12
making data block
writing data block 47 to disk
enter name: parent=(3 2) name=dir1
enter name: dir1 need_len=12
parent data blk[0] = 33
step to LAST entry in data block 33
. .. lost+found
found space: name=lost+found ideal=20 rlen=1000 remain=980
write parent data block[0]=33
----------------------------------------
input command : [ls|cd|pwd|mkdir|creat|quit] ls
i_block[0] = 33
drwxr-xr-x   4   0   0  Oct 22 10:32    1024    .
drwxr-xr-x   4   0   0  Oct 22 10:32    1024    ..
drwx------   2   0   0  Oct 22 10:32   12288    lost+found
drwxr-xr-x   2   0   0  Oct 22 10:33    1024    dir1
i_block[1] = 0
input command : [ls|cd|pwd|mkdir|creat|quit]
```

Fig. 11.9 Sample outputs of project #2

11.8.4 Algorithm of rmdir

```
/************ Algorithm of rmdir *************/
  (1). get in-memory INODE of pathname:
         ino = getino(pathanme);
         mip = iget(dev, ino);

  (2). verify INODE is a DIR (by INODE.i_mode field);
         minode is not BUSY (refCount = 1);
         verify DIR is empty (traverse data blocks for number of entries = 2);

  (3). /* get parent's ino and inode */
         pino = findino();  //get pino from .. entry in INODE.i_block[0]
         pmip = iget(mip->dev, pino);

  (4). /* get name from parent DIR's data block
         findname(pmip, ino, name); //find name from parent DIR
```

```
(5).  remove name from parent directory */
      rm_child(pmip, name);

(6). dec parent links_count by 1; mark parent pimp dirty;
      iput(pmip);

(7). /* deallocate its data blocks and inode */
      bdalloc(mip->dev, mip->INODE.i_blok[0];
      idalloc(mip->dev, mip->ino);
      iput(mip);
```

In the rmdir algorithm, most steps are simple and self-explanatory. We only need to explain the following steps in more detail.

1. **How to test DIR empty:** Every DIR's links_count starts with 2 (for the . and .. entries). Each subdirectory increments its links_count by 1 but regular files do not increment the links_count of the parent directory. So, if a DIR's links_count is greater than 2, it is definitely not empty. However, if links_count = 2, the DIR may still contain regular files. In that case, we must traverse the DIR's data blocks to count the number of dir_entries, which must be greater than 2. This can be done by the same technique of stepping through dir_entris in the data blocks of a DIR INODE.

2. **How to deallocate inode and data block**: When removing a DIR, we must deallocate its inode and data block (numbers). The function

   idalloc(dev, ino)

   deallocates an inode (number). It clears the ino's bit in the device's inodes bitmap to 0. Then it increments the free inodes count in both superblock and group descriptor by 1.

```
int clr_bit(char *buf, int bit) // clear bit in char buf[BLKSIZE]
{   buf[bit/8] &= ~(1 << (bit%8)); }

int incFreeInodes(int dev)
{
  char buf[BLKSIZE];
  // inc free inodes count in SUPER and GD
  get_block(dev, 1, buf);
  sp = (SUPER *)buf;
  sp->s_free_inodes_count++;
  put_block(dev, 1, buf);
  get_block(dev, 2, buf);
  gp = (GD *)buf;
  gp->bg_free_inodes_count++;
  put_block(dev, 2, buf);
}
int idalloc(int dev, int ino)
{
  int i;
  char buf[BLKSIZE];
  MTABLE *mp = (MTABLE *)get_mtable(dev);
```

```
    if (ino > mp->ninodes){ // niodes global
       printf("inumber %d out of range\n", ino);
       return;
    }
    // get inode bitmap block
    get_block(dev, mp->imap, buf);
    clr_bit(buf, ino-1);
    // write buf back
    put_block(dev, mp->imap, buf);
    // update free inode count in SUPER and GD
    incFreeInodes(dev);
}
```

Deallocating a disk block number is similar, except it uses the device's blocks bitmap and it increaments the free blocks count in both superblock and groupdescriptor. This is left as an exercise.

Exercise 11.11 Implement a **bdalloc(int dev, int bno)** function which deallocates a disk block (number) bno.

3. **Remove dir_entry form parent DIR**: The function

```
rm_child(MINODE *pmip, char *name)
```

removes the dir_entry of name from a parent directory minode pointed by pmip. The algorithm of rm_child is as follows.

```
/*************** Algorithm of rm_child ***************/
(1). Search parent INODE's data block(s) for the entry of name
(2). Delete name entry from parent directory by
(2).1. if (first and only entry in a data block){
       In this case, the data block looks like the following diagram.

       ------------------------------------------
       |ino rlen=BLKSIZE nlen NAME               |
       ------------------------------------------

       deallocate the data block; reduce parent's file size by BLKSIZE;
       compact parent's i_block[ ] array to eliminate the deleted entry if it's
       between nonzero entries.
           }
(2).2. else if LAST entry in block{
       Absorb its rec_len to the predecessor entry, as shown in the following diagrams.

        Before                        |remove this entry   |
        ------------------------------------------------------
        xxxxx|INO rlen nlen NAME |yyy  |zzz rec_len         |
        ------------------------------------------------------
        After
        ------------------------------------------------------
```

```
        xxxxx|INO rlen nlen NAME |yyy (add rec_len to yyy)  |
        ------------------------------------------------
           }
(2).3. else: entry is first but not the only entry or in the middle of a block:
          {
            move all trailing entries LEFT to overlay the deleted entry;
            add deleted rec_len to the LAST entry; do not change parent's file
            size;

            The following diagrams illustrate the block contents before and after
            deleting such an entry.

            Before:  | delete this entry |<= move these LEFT  |
                ---------------------------------------------
                xxxxx|ino rlen nlen NAME |yyy|...|zzz         |
                -----|-------------------|------ size ---------
                     dp                  cp
            After:
                -----|---- after move LEFT -------------------
                xxxxx|yyy|...|zzz (rec_len += rlen)           |
                ---------------------------------------------

            How to move trailing entries LEFT? Hint: memcpy(dp, cp, size);
          }
```

11.8.5 Implementation of rmdir

The programming task here is to implement the rmdir function and add the rmdir command to the file system. Compile and run the resulting program to demonstrate rmdir operation. Figure 11.10 shows the sample outputs of the resulting file system. The figure shows the detailed steps of removing a directory.

4. **link:** The command

```
link old_file  new_file
```

creates a hard link from new_file to old_file. Hard links can only be applied to regular files, not to DIRs, because linking to DIRs may create loops in the file system name space. Hard link files share the same inode. Therefore, they must be on the same device. The algorithm of link is as follows.

11.8.6 Algorithm of link

```
/******************** Algorithm of link ******************/
(1). // verify old_file exists and is not a DIR;
        oino = getino(old_file);
        omip = iget(dev, oino);
        check omip->INODE file type (must not be DIR).
```

```
fs_init()
mount_root
checking EXT2 FS : OK
bmp = 8 imap = 9 iblock = 10
mydisk mounted on / OK
root refCount = 3
input command : [ls|cd|pwd|mkdir|creat|rmdir|quit] ls
i_block[0] = 33
drwxr-xr-x   5   0   0  Oct 22 10:39    1024    .
drwxr-xr-x   5   0   0  Oct 22 10:39    1024    ..
drwx------   2   0   0  Oct 22 10:39   12288    lost+found
drwxr-xr-x   2   0   0  Oct 22 10:39    1024    dir1
drwxr-xr-x   2   0   0  Oct 22 10:39    1024    dir2
i_block[1] = 0
input command : [ls|cd|pwd|mkdir|creat|rmdir|quit] rmdir dir1
getino: pathname=dir1
tokenize dir1
dir1
========================================
getino: i=0 name[0]=dir1
search for dir1 in MINODE = [3, 2]
search: i=0  i_block[0]=33
   i_number rec_len name_len    name
       2      12       1        .
       2      12       2        ..
      11      20      10        lost+found
      12      12       4        dir1
found dir1 : ino = 12
[3 12] refCount=1
last entry=[13 968] last entry=[13 980]
input command : [ls|cd|pwd|mkdir|creat|rmdir|quit] ls
i_block[0] = 33
drwxr-xr-x   4   0   0  Oct 22 10:39    1024    .
drwxr-xr-x   4   0   0  Oct 22 10:39    1024    ..
drwx------   2   0   0  Oct 22 10:39   12288    lost+found
drwxr-xr-x   2   0   0  Oct 22 10:39    1024    dir2
i_block[1] = 0
input command : [ls|cd|pwd|mkdir|creat|rmdir|quit]
```

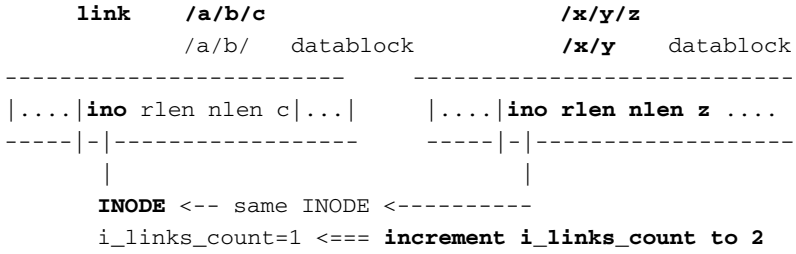
Fig. 11.10 Sample outputs of project #3

```
(2). // new_file must not exist yet:
        getino(new_file) must return 0;
(3). creat new_file with the same inode number of old_file:
        parent = dirname(new_file);  child = basename(new_file);
        pino = getino(parent);
        pmip = iget(dev, pino);
        // creat entry in new parent DIR with same inode number of old_file
        enter_name(pmip, oino, child);
(4). omip->INODE.i_links_count++;  // inc INODE's links_count by 1
     omip->dirty = 1;              // for write back by iput(omip)
     iput(omip);
     iput(pmip);
```

We illustrate the link operation by an example. Consider.

link /a/b/c /x/y/z

The following diagrams show the outcome of the link operation. It adds an entry z to the data block of /x/y. The inode number of z is the same ino of a/b/c, so they share the same INODE, whose links_count is incremented by 1.



5. **unlink:** The command

```
unlink filename
```

unlinks a file. It decrements the file's links_count by 1 and deletes the file name from its parent DIR. When a file's links_count reaches 0, the file is truly removed by deallocating its data blocks and inode. The algorithm of unlink is

11.8.7 Algorithm of unlink

```
/**************  Algorithm of unlink *************/
(1). get filenmae's minode:
        ino = getino(filename);
        mip = iget(dev, ino);
        check it's a REG or symbolic LNK file; can not be a DIR
(2). // remove name entry from parent DIR's data block:
        parent = dirname(filename); child = basename(filename);
        pino = getino(parent);
        pimp = iget(dev, pino);
        rm_child(pmip, ino, child);
        pmip->dirty = 1;
        iput(pmip);
(3). // decrement INODE's link_count by 1
         mip->INODE.i_links_count--;
(4).     if (mip->INODE.i_links_count > 0)
             mip->dirty = 1;  // for write INODE back to disk
(5).     else{  // if links_count = 0: remove filename
             deallocate all data blocks in INODE;
             deallocate INODE;
         }
         iput(mip);           // release mip
```

6. **symlink:** The command

```
symlink old_file  new_file
```

creates a symbolic link from new_file to old_file. Unlike hard links, symlink can link to anything, including DIRs, even files not on the same device. The algorithm of symlink is

11.8.8 Algorithm of symlink

```
/********* Algorithm of symlink old_file new_file *********/
(1). check: old_file must exist and new_file not yet exist;
(2). creat new_file; change new_file to LNK type;
(3). // assume length of old_file name <= 60 chars
        store old_file name in newfile's INODE.i_block[ ] area.
        set file size to length of old_file name
        mark new_file's minode dirty;
        iput(new_file's minode);
(4). mark new_file parent minode dirty;
     iput(new_file's parent minode);
```

7. readlink: The function

```
int readlink(file, buffer)
```

reads the target file name of a symbolic file and returns the length of the target file name. The algorithm of readlink() is.

11.8.9 Algorithm of readlink

```
/************* Algorithm of readlink (file, buffer) *************/
(1). get file's INODE in memory; verify it's a LNK file
(2). copy target filename from INODE.i_block[ ] into buffer;
(3). return file size;
```

11.8.10 Other Level-1 Functions

Other level-1 functions include access, chmod, chown, change file's time fields, etc. The operations of all such functions are of the same pattern:

```
(1). get the in-memory INODE of a file by
       ino = getino(pathname);
       mip = iget(dev,ino);
(2). get information from INODE or modify the INODE;
(3). if INODE is modified, set mip->dirty to zonzero for write back;
(4). iput(mip);
```

```
root@wang:~/abc/360/F17/TEST4# a.out
checking EXT2 FS ....OK
bmp=8 imap=9 inode_start = 10
init()
mount_root()
mydisk mounted on / OK
creating P0 as running process
input command: [ls|cd|pwd|mkdir|creat|rmdir|link|unlink|symlink|readlink|chmod|utime|quit] ls
cmd=ls path= param=
i_block[0] = 33
drwxr-xr-x   3   0   0  Apr 21 19:50    1024    .
drwxr-xr-x   3   0   0  Apr 21 19:50    1024    ..
drwx------   2   0   0  Apr 21 19:50   12288    lost+found
drwxr-xr-x   2   0   0  Apr 21 19:50    1024    dir2
drwxr-xr-x   2   0   0  Apr 26 16:03    1024    dir1
-rw-r--r--   1   0   0  Apr 26 16:03       0    file1
lrwxrwxrwx   1   0   0  Dec 23 07:18       4    hi -> dir1
i_block[1] = 0
input command: [ls|cd|pwd|mkdir|creat|rmdir|link|unlink|symlink|readlink|chmod|utime|quit]
```

Fig. 11.11 Sample outputs of file system level-1

Examples of Other Level-1 Operations

1. **chmod oct filename:** Change filename's permission bits to octal value
2. **utime filename**: change file's access time to current time:

11.8.11 Programming Project #1: Implementation of File System Level-1

The programming project #1 is to complete the level-1 implementation of the file system and demonstrate the file system. Figure 11.11 shows the sample outputs of running the Level 1 implementation of the file system. The ls command shows that hi is a symbolic link to the directory dir1.

11.9 File System Level-2 Functions

Level-2 of the file system implements read and write operations of file contents. It consists of the following functions: open, close, lseek, read, write, opendir and readdir.

1. Open Operation
In Unix/Linux, the **system call**

```
int open(char *filename, int flags);
```

opens a file for read or write, where flags is one of O_RDONLY, O_WRONLY, O_RDWR, which may be bitwise or-ed with O_CREAT, O_APPEND, O_TRUNC, etc. These symbolic constants are defined in the **fcntl.h** file. On success, open() returns a file descriptor (number), which is used in subsequent system calls, such as read(), write(), lseek() and close(), etc. For simplicity, we shall assume the parameter flags = 0|1|2|3 or RD|WR|RW|AP for READ|WRITE| RDWR|APEND, respectively. The algorithm of open() is.

11.9.1 Algorithm of open

```
/******************  Algorithm of open ********************/
(1). get file's minode:
       ino = getino(filename);
       if (ino==0){  // if file does not exist
           creat(filename);          // creat it first, then
           ino = getino(filename);   // get its ino
       }
       mip = iget(dev, ino);

(2). allocate an openTable entry OFT; initialize OFT entries:
       mode = 0(RD) or 1(WR) or 2(RW) or 3(APPEND)
       minodePtr = mip;              // point to file's minode
       refCount = 1;
       set offset = 0 for RD|WR|RW; set to file size for APPEND mode;

(3). Search for the first FREE fd[index] entry with the lowest index in PROC;
       fd[index] = &OFT;          // fd entry points to the openTable entry;

(4). return index as file descriptor;
```

Figure 11.12 shows the data structure created by open. In the figure, (1) is the PROC structure of the process that calls open(). The returned file descriptor, fd, is the index of the fd[ ] array in the PROC structure. The contents of fd[fd] points to an OFT, which points to the minode of the file. The OFT's refCount represents the number of processes that share the same instance of an opened file. When a process opens a file, the refCount in the OFT is set to 1. When a process forks a child process, the child process inherits all the opened file descriptors of the parent, which increments the refCount of every shared OFT by 1. When a process closes a file descriptor, it decrements OFT.refCount by 1. When OFT.refCount reaches 0, the file's minode is released and the OFT is deallocated. The OFT's offset is a conceptual pointer to the current byte position in the file for read/write. It is initialized to 0 for RD|WR|RW mode or to file size for APPEND mode.

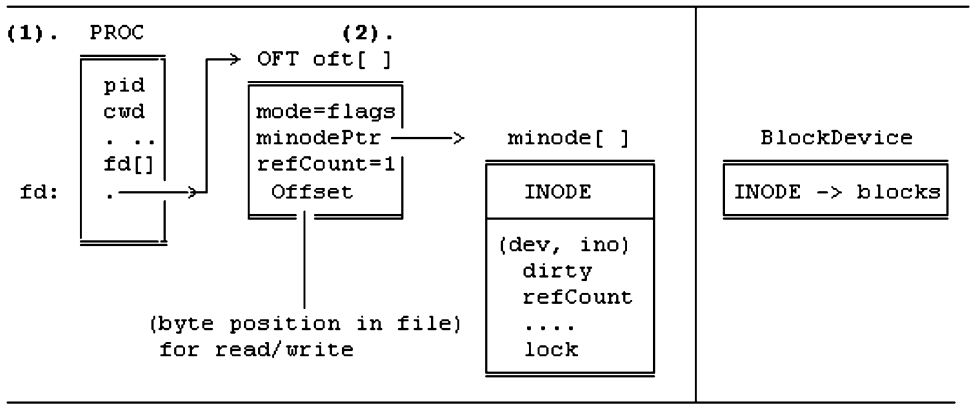


Fig. 11.12 Data structures of open

11.9.2 lseek

In Linux, the system call.

```
lseek(fd, position, whence); // whence=SEEK_SET or SEEK_CUR
```

sets the offset in the OFT of an opened file descriptor to the byte **position** either from the file beginning (**SEEK_SET**) or relative to the current position (**SEEK_CUR**). For simplicity, we shall assume that the new position is always from the file beginning. Once set, the next read/write begins from the current offset position. The algorithm of lseek is trivial. It only needs to check the requested position value is within the bounds of [0, fileSize-1]. We leave the implementation of lseek as an exercise.

Exercise 11.12 Write C code for the lseek function **int lseek(int fd, int position**).

2. Close Operation
The **close(int fd)** operation closes a file descriptor. The algorithm of close is

11.9.3 Algorithm of close

```
/***************  Algorithm of close  ***************/
(1). check fd is a valid opened file descriptor;
(2). if (PROC's fd[fd] != 0){       // points to an OFT
   .     OFT.refCount--;            // dec OFT's refCount by 1
         if (refCount == 0)         // if last process using this OFT
             iput(OFT.minodePtr);   // release minode
     }
(4). PROC.fd[fd] = 0;              // clear PROC's fd[fd] to 0
```

11.9.4 Read Regular Files

In Unix/Linux, the system call

```
int read(int fd, char *buf, int nbytes);
```

reads nbytes from an opened file descriptor into a buffer area in user space. The read system call is routed to the read function in the OS kernel. For regular files, the algorithm of read is.

```
/********* Algorithm of read(int fd,  char *buf,  int nbytes) *********/
(1). count = 0;              // number of bytes read
     offset = OFT.offset;    // byte offset in file to READ
     compute bytes available in file: avil = fileSize – offset;
(2). while (nbytes && avil){
         compute logical block: lbk   = offset  /  BLKSIZE;
         start byte in block:       start = offset % BLKSIZE;
(3).     convert logical block number, lbk, to physical block number, blk,
         through INODE.i_block[ ] array;
```

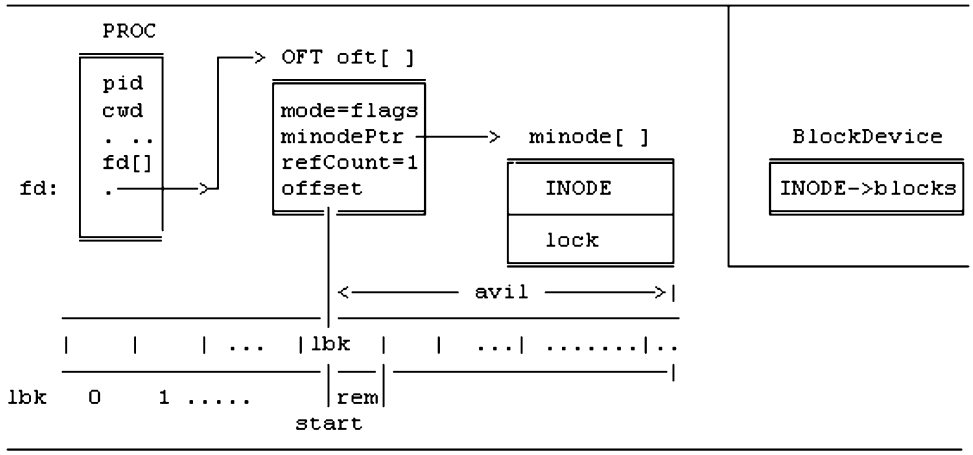


Fig. 11.13 Data structures for read

```
(4).      get_block(dev, blk, kbuf);   // read blk into char kbuf[BLKSIZE];
          char *cp = kbuf + start;
          remain = BLKSIZE - start;
(5).      while (remain){   // copy bytes from kbuf[ ] to buf[ ]
             *buf++ = *cp++;
             offset++; count++;           // inc offset, count;
             remain--; avil--; nbytes--;  // dec remain, avail, nbytes;
             if (nbytes==0 || avil==0)
                 break;
          } // end of while(remain)
      }     //  end of while(nbytes && avil)
(6). return count;
```

The algorithm of read() can be best explained in terms of Fig. 11.13. Assume that fd is opened for READ. The offset in the OFT points to the current byte position in the file from where we wish to read nbytes. To the file system in kernel, a file is just a sequence of contiguous bytes, numbered from 0 to fileSize-1. As Fig. 11.13 shows, the current byte position, offset, falls in a logical block.

```
lbk = offset  /  BLKSIZE,
```

the byte to start read is.

```
start = offset % BLKSIZE
```

and the number of bytes remaining in the logical block is.

```
remain = BLKSIZE - start.
```

At this moment, the file has.

```
avil = fileSize - offset
```

bytes still available for read. These numbers are used in the read algorithm.

For small EXT2 files systems, the block size is 1 KB and files have at most double indirect blocks. For read, the algorithm of converting logical blocks to physical blocks is.

```
/* Algorithm of Converting Logical Blocks to Physical Blocks */
  int map(INODE, lbk){                  // convert lbk to blk via INODE
       if (lbk < 12)                    // direct blocks
           blk = INODE.i_block[lbk];
       else if (12 <= lbk < 12+256){    // indirect blocks
           read INODE.i_block[12] into int ibuf[256];
           blk = ibuf[lbk-12];
       }
       else{   // doube indirect blocks; see Exercise 11.13 below.
        }
        return blk;
  }
```

Exercise 11.13 Complete the algorithm for converting double indirect logical blocks to physical blocks. Hint: Mailman's algorithm.

Exercise 11.14 For simplicity and clarity, step (5) of the read algorithm transfers data one byte at a time and updates the control variable on each byte transfer, which are not very efficient. Optimize the code by transferring a maximal sized chunk of data at a time.

11.9.5 Write Regular Files

In Unix/Linux, the system call

```
int write(int fd, char buf[ ], int nbytes);
```

writes nbytes from buf in user space to an opened file descriptor and returns the actual number of bytes written. The write system call is routed to the write function in the OS kernel. For regular files, the algorithm of write is

```
/******* Algorithm of write(int fd, char *buf, int nbytes) *******/
(1). count = 0;          // number of bytes written
(2). while (nbytes){
       compute logical block: lbk = oftp->offset  / BLOCK_SIZE;
       compute start byte:     start = oftp->offset % BLOCK_SIZE;
(3).   convert lbk to physical block number, blk, through the i_block[ ] array;
(4).   read_block(dev, blk, kbuf); // read blk into kbuf[BLKSIZE];
       char *cp = kbuf + start;  remain = BLKSIZE - start;
(5)    while (remain){                // copy bytes from buf[ ] to kbuf[ ]
          *cp++ = *buf++;
          offset++;  count++;        // inc offset, count;
          remain --; nbytes--;       // dec remain, nbytes;
          if (offset > fileSize)  fileSize++; // inc file size
          if (nbytes <= 0) break;
```

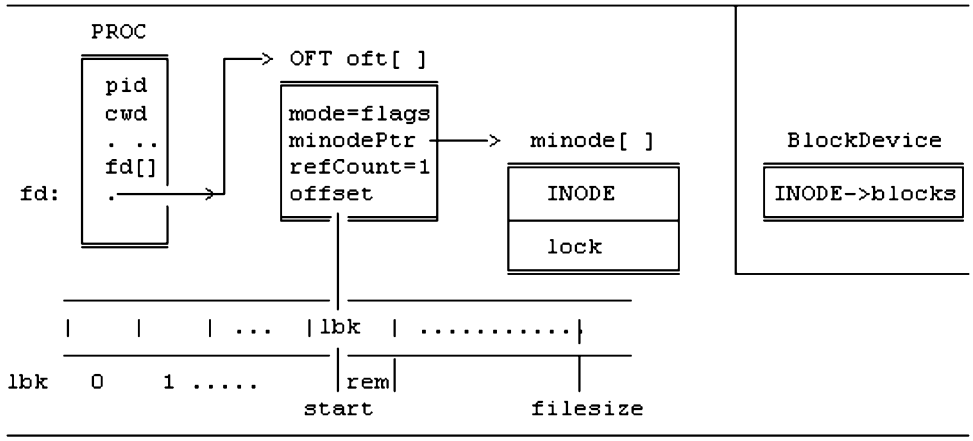


Fig. 11.14 Data structures for write

```
        } // end while(remain)
(6).    write_block(dev, blk, kbuf);
    } // end while(nbytes)
(7). set minode dirty = 1;  // mark minode dirty for iput() when fd is closed
    return count;
```

The algorithm of write can be best explained in terms of Fig. 11.14. In the figure, the offset in the OFT is the current byte position in the file to write, which falls in a logical block. It uses offset to compute the logical block number, lbk, the start byte position and the number of bytes remaining in the logical block. It then converts the logical block to physical block through the file's INODE.i_block array.Then it reads the physical block into a kbuf and transfer data from buf into kbuf, which may increase the file size if offset exceeds the current file size. After transferring data to kbuf, it writes the buffer back to disk. The write algorithm stops when all nbytes are written. The return value is nbtyes unless write fails.

The algorithm of converting logical block to physical block for write is similar to that of read, except for the following difference. During write, the intended data block may not exist. If a direct block does not exist, it must be allocated and recorded in the INODE. If the indirect block i_block [12] does not exist, it must be allocated and initialized to 0. If an indirect data block does not exist, it must be allocated and recorded in the indirect block. Simliarly, if the double indirect block i_block [13] dose not exist, it must be allocated and initialized to 0. If an entry in the double indirect block does not exist, it must be allocated and initialized to zero. If a double indirect block does not exist, it must be allocated and recorded in a double indirect block entry, etc.

Exercise 11.15 In the write algorithm, a file opened for WRITE mode may start with no data blocks at all. Complete the algorithm for coverting logical block numbers to physical blocks, including direct, indirect and double indirect blocks.

Exercise 11.16 For simplicity and clarity, step (5) of the write algorithm transfers data one byte at a time and updates the control variable on each byte transfer. Optimize the code by transferring a maximal sized chunk of data at a time.

Exercise 11.17 With open() and read(), implement the command cat filename, which displays the contents of a file.

Exercise 11.18 With open(), read() and write(), implement the command cp src dest, which copies a src file src to a dest file.

Exercise 11.19 Implement the command function mv file1 file2.

11.9.6 Opendir-Readdir

Unix considers everything as a file. Therefore, we should be able to open a DIR for read just like a regular file. From a technical point of view, there is no need for a separate set of opendir() and readdir() functions. However, different Unix systems may have different file systems. It may be difficult for users to interpret the contents of a DIR file. For this reason, POSIX specifies **opendir** and **readdir** operations, which are independent of file systems. Support for opendir is trivial; it's the same open system call, but readdir() has the form.

```
struct dirent *ep = readdir(DIR *dp);
```

which returns a pointer to a dirent structure on each call. This can be implemented in user space as a library I/O function. Instead, we shall implement opendir() and readir() directly.

```
int opendir(pathaname)
{   return open(pathname, RD|O_DIR); }
```

where O_DIR is a bit pattern for opening the file as a DIR. In the open file table, the mode field contains the O_DIR bit.

```
int readdir(int fd, struct udir *udirp) // struct udir{DIR udir;};
{
   // same as read() except:
   use the current byte offset in OFT to read the next dir_entry;
   copy the dir_entry into *udirp;
   advance offset by dir_entry's rec_len for reading next dir_entry;
}
```

11.9.7 Programming Project #2: Implementation of File System Level-2

The programming project #2 is to complete the implementation of File System Level-2 and demonstrate read, write, cat, cp and mv operations in the file system. The reader may consult earlier Chapters for cat and cp operations. For the mv operation, see Problem 12. Figure 11.15 show the sample outputs of running the complete Level 2 file system.

```
root@wang:~/abc/360/F17/TEST5# a.out
checking EXT2 FS ....OK
bmp=8 imap=9 inode_start = 10
init()
mount_root()
root refCount = 1
creating P0 as running process
root refCount = 2
input command: [ls|cd|pwd|mkdir|creat|rmdir|link|symlink|unlink
               |open|close|lseek|read|write|cat|cp|mv|quit] : ls
cmd=ls path= param=
i_block[0] = 33
drwxr-xr-x   3   0   0  Nov  9 12:54    1024    .
drwxr-xr-x   3   0   0  Nov  9 12:54    1024    ..
drwx------   2   0   0  Nov  3 09:07   12288    lost+found
-rw-r--r--   1   0   0  Nov  9 12:54      42    f1
drwxr-xr-x   2   0   0  Apr 21 18:49    1024    dir2
-rw-r--r--   1   0   0  Apr 10 08:48       0    file2
i_block[1] = 0
input command: [ls|cd|pwd|mkdir|creat|rmdir|link|symlink|unlink
               |open|close|lseek|read|write|cat|cp|mv|quit] :
```

Fig. 11.15 File system level-2

11.10 File System Level-3

File system level-3 supports mount, umount of file systems and file protection.

Mount Operation
The command

```
mount filesys mount_point
```

mounts a file system to a mount_point directory. It allows the file system to include other file systems as parts of the existing file system. The data structures used in mount are the MOUNT table and the in-memory minode of the mount_point directory. The algorithm of mount is

11.10.1 Algorithm of mount

```
/********* Algorithm of mount **********/
 1. If no parameter, display current mounted file systems;
 2. Check whether filesys is already mounted:
    The MOUNT table entries contain mounted file system (device) names
    and their mounting points. Reject if the device is already mounted.
    If not, allocate a free MOUNT table entry.
 3. Open the filesys virtual disk (under Linux) for RW; use (Linux) file
    descriptor as new dev. Read filesys superblock to verify it is an EXT2 FS.
```

```
4. Find the ino, and then the minode of mount_point:
        ino = getino(pathname);    // get ino:
        mip = iget(dev, ino);      // load its inode into memory;
5. Check mount_point is a DIR and not busy, e.g. not someone's CWD.
6. Record new dev and filesys name in the MOUNT table entry, store its
   ninodes, nblocks, bmap, imap and inodes start block, etc. for quick
   access.
7. Mark mount_point minode as mounted on (mounted flag = 1) and let it point
   at the MOUNT table entry, which points back to the mount_point minode.
```

Umount Operation:
The operation **umount filesys**
un-mounts a mounted file system. It detaches a mounted file system from the mounting point, where filesys may be a virtual diak name or a mounting point directory name. The algorithm of umount is

11.10.2 Algorithm of umount

```
/****************** Algorithm of umount ***********************/
  1. Search the MOUNT table to check filesys is indeed mounted.
  2. Check whether any file is active in the mounted filesys; If so, reject;
  3. Find the mount_point in-memory inode, which should be in memory while
     it's mounted on. Reset the minode's mounted flag to 0; then iput() the
     minode.
```

To check whether a mounted file system contains any files that are actively in use, search the in-memory minodes for any entry whose dev matches the dev number of the mounted file system.

11.10.3 Cross Mounting Points

While it is easy to implement mount and umount, there are implications. With mount, we must modify the getino(pathname) function to support crossing mount points. Assume that a file system, newfs, has been mounted on the directory /a/b/c/. When traversing a pathname, mount point crossing may occur in both directions.

(1) **Downward traversal:** When traversing the pathname /a/b/c/x, once we reach the minode of /a/b/c, we should see that the minode has been mounted on (mounted flag=1). Instead of searching for x in the INODE of /a/b/c, we must

- Follow the minode's mntPtr pointer to locate the mount table entry.
- From the mount table's dev number, get its root (ino=2) INODE into memory.
- Then continue search for x under the root INODE of the mounted device.

(2) **Upward traversal:** Assume that we are at the directory /a/b/c/x/ and traversing upward, e.g. cd ../../, which will cross the mount point /a/b/c. When we reach the root INODE of the mounted file system, we should see that it is a root directory (ino=2) but its dev number differs from that of the

real root, so it is not the real root yet. Using its dev number, we can locate its mount table entry, which points to the mounted minode of /a/b/c/. Then, we switch to the minode of /a/b/c/ and continue the upward traversal. Thus, crossing mount point is like a monkey or squirrel hoping from one tree to another and then back.

Since crossing mounting points changes the device number, a single global device number becomes inadequate.We must modify the getino() function as

```
int getino(char *pathname, int *dev)
```

and modify calls to getino() as

```
int dev;                    // local in function that calls getino()
if (pathnme[0]=='/')
   dev = root->dev;         // absolute pathname
else
   dev = running->cwd->dev; // relative pathname

int ino = getino(pathname, &dev); // pass &dev as an extra parameter
```

In the modified getino() function, change *dev whenever the pathname crosses a mounting point. Thus, the modified getino() function essentially returns (**dev**, ino) of a pathname with **dev** being the final device number.

11.10.4 File Protection

In Unix/Linux, file protection is by permission bits in the file's INODE. Each file's INODE has an i_mode field, in which the low 9 bits are permissions. The 9 permission bits are

```
owner     group    other
-------   -------  -------
r w x     r w x    r w x
```

The first 3 bits apply to the owner of the file, the second 3 bits apply to users in the same group as the owner, and the last 3 bits apply to all others. For directories, the x bit indicates whether a process is allowed to go into the directory. Each process has a uid and a gid. When a process tries to access a file, the file system checks the process uid and gid against the file's permission bits to determine whether it is allowed to access the file with the intended mode of operation. If the process does not have the right permission, the access will be denied. For the sake of simplicity, we may ignore process gid and use only the process uid to check access permission.

11.10.5 Real and Effective uid

In Unix/Linux, each process has a **real uid** and an **effective uid**. The file system checks the access rights of a process by its effective uid. Under normal conditions, the effective uid and real uid of a process are identical. When a process executes a **setuid** program, which has the **setuid** bit in the file's

i_mode field turned on, the process' effective uid becomes the uid of the program. While executing a **setuid program**, the process effectively becomes the owner of the program file. For example, when a process executes the mail program, which is a setuid program owned by the superuser, it can write to the mail file of another user. When a process finishes executing a setuid program, it reverts back to the real uid. For simplicity, we shall ignore effective uid also.

11.10.6 File Locking

File locking is a mechanism which allows a process to set locks on a file, or parts of a file to prevent race conditions when updating files. File locks can be either shared, which allows concurrent reads, or exclusive, which enforces exclusive write. File locks can also be mandatory or advisory. For example, Linux supports both shared and exclusive files locks but file locking is only advisory. In Linux, file locks can be set by the fcntl() system call and manipulated by the flock() system call. For simplicity, we shall assume only a very simple kind of file locking. When a process tries to open a file, the intended mode of operation is checked for compatibility. The only compatible modes are READs. If a file is already opened for updating mode, i.e. WR|RW|APPEND, it cannot be opened again. However, this does not prevent related processes, e.g. a parent process and a child process, from modifying the same file that's opened by the parent, but this is also true in Unix/Linux. In that case, the file system can only guarantee each write operation is atomic but not the writing order of the processes, which depends on process scheduling.

11.10.7 Programming Project #3: Implementation of Complete File System

Programming Project #3 is to complete the file system implementation by including functions in the Level-3 of the file system. Sample solution of the file system is available for download at the book's website. Source code of the complete file system is available for instructors upon request.

11.11 Extensions of File System Project

The simple EXT2 file system uses 1 KB block size and has only 1 disk block group. It can be extended easily as follows.

(1) Multiple groups: The size of group descriptor is 32 bytes. With 1 KB block size, a block may contain 1024/32 = 32 group descriptors. With 32 groups, the file system size can be extended to 32*8 = 256 MB.
(2) 4 KB block size: With 4 KB block size and only one group, the file system size would be 4*8 = 32 MB. With one group descriptor block, the file system may have 128 groups, which extend the file system size to 128*32 = 4GB. With 2 group descriptor blocks, the file system size would be 8GB, etc. Most of the extensions are straightforward, which are suitable topics for programming projects.
(3) Pipe files: It's possible to implement pipes as regular files, which obey the read/write protocol of pipes. The advantages of this scheme are: it unifies pipes and file inodes and it allows named pipes, which can be used by unrelated processes. In order to support fast read/write operations, pipe contents should be in memory, such as a RAMdisk. If desired, the reader may implement named pipes as FIFO files.

(4) I/O Buffering: In the Programming Project, every disk block is read in or written out directly. This would incur too much physical disk I/O operations. For better efficiency, a real file system usually uses a set of I/O buffers as a cache memory for disk blocks. File system I/O buffering is covered in Chap. 12, but it may be incorporated into the file system project.

11.12 Summary

This chapter covers EXT2 file system. The goal of this chapter is to lead the reader to implement a complete EXT2 file system that is totally Linux compatible. The premise is that if the reader understands one file system well, it should be easy to adapt to any other file systems. It first describes the historic role of EXT2 file system in Linux and the current status of EXT3/EXT4 file systems. It uses programming examples to show the various EXT2 data structures and how to traverse the EXT2 files system tree. Then it shows how to implement an EXT2 file system which supports all file operations as in the Linux kernel. It shows how to build a base file system by mount_root from a virtual disk. Then it divides the file system implementation into 3 levels. Level-1 expands the base file system to implement the file system tree. Level-2 implements read/write operations of file contents. Level-3 implements mount/umount of file systems and file protection. In each level, it describes the algorithms of the file system functions and demonstrates their implementations by programming examples. Each level is cumulated by a programming project. The final project is to integrate all the programming examples and exercises into a fully functional file system. The programming project has been used as the class project in the CS360, Systems Programming, course at EECS, WSU for many years, and it has proved to be very effective and successful.

Problems

1. Write a C program to display the group descriptor of an EXT2 file system on a device.
2. Modify the imap.c program to print the inodes bitmap in char map form, i.e. for each bit, print a '0' if the bit is 0, print a '1' if the bit is 1.
3. Write a C program to display the blocks bitmap of an Ext2 file system, also in char map form.
4. Write a C program to print the dir_entries of a directory.
5. A virtual disk can be mounted under Linux as a loop device, as in

```
sudo mount -o loop mydisk /mnt     # mount mydisk to /mnt
mkdir /mnt/newdir                  # create new dirs and copy files to /mnt
sudo umount /mnt                   # umount when finished
```

   Run the dir.c program on mydisk again to verify the newly created dirs and files still exist.
6. Given an INODE pointer to a DIRectory inode. Write a

```
int  search(INODE *dir, char *name)
```

   function which searches for a dir_entry with a given name. Return its inode number if found, else return 0.
7. Given a device containing an EXT2 file system and a pathname, .e.g. /a/b/c/d, write a C function

```
INODE *path2inode(int fd, char *pathname)  // assume fd=file descriptor
```

   which returns an INODE pointer to the file's inode, or 0 if the file in inaccessible.

8. Assume: a small EXT2 file system with 1 KB block size and only one blocks group.

    1. Prove that a file may only have some double-indirect blocks but no triple-indirect blocks.
    2. Given the pathname of a file, e.g. /a/b/c/d, write a C program, showblock, which prints all disk block numbers of the file. Hint: use Mailman's algorithm for double indirect blocks, if any.

9. In the Programming Project #1, an in-memory minode is released as FREE when its refCount reaches 0. Design and implement a scheme, which uses the minode[ ] area as a cache memory for in-memory INODES. Once an INODE is loaded into a minode slot, try to keep it in memory for as long as possible even if no process is actively using it. Whenever a process finds a needed INODE in memory, count it as a cache hit. Otherwise, it's a cache miss. Modify the iget(dev, ino) function to count the number of cache hits.
10. Implement step (2) of the pwd algorithm as a utility function

```
int get_myino(MINODE *mip, int *parent_ino)
```

    which returns the inode number of . and that of .. in parent_ino.
11. Implement step (4) of the pwd algorithm as a utility function

```
int get_myname(MINODE *parent_minode, int my_ino, char *my_name)
```

    which returns the name string of a dir_entry identified by my_ino in the parent directory.
12. Implement the **mv file1 file2** operation, which moves file1 to file2. For files on the same device, mv should just rename file1 as file2 without copying file contents. Hints: link-unlink. How to implement mv if the files are on different devices?

References

Card, R., Theodore Ts'o, T., Stephen Tweedie, S., "Design and Implementation of the Second Extended Filesystem", web.mit.edu/tytso/www/linux/ext2intro.html, 1995

Cao, M., Bhattacharya, S, Tso, T., "Ext4: The Next Generation of Ext2/3 File system", IBM Linux Technology Center, 2007.

EXT2: http://www.nongnu.org/ext2-doc/ext2.html, 2001

EXT3: http://jamesthornton.com/hotlist/linux-filesystems/ext3-journal, 2015