7 File Operations

Abstract

This chapter covers file systems. It explains the various levels of file operations in operating systems. These include preparing storage devices for file storage, file system support functions in kernel, system calls, library I/O functions on file streams, user commands and sh scripts for file operations. It presents a systematic overview of file operations, from read/write file streams in user space to system calls to kernel space down to the device I/O driver level. It describes low level file operations, which include disk partitions, example programs to display partition tables, format partitions for file systems and mount disk partitions. It presents an introduction to the EXT2 file system of Linux, which include the EXT2 file system data structures, example programs to display superblock, group descriptor, blocks and inodes bitmaps and directory contents. The programming project is to integrate the knowledge about EXT2/3 file systems and programming techniques presented in the chapter into a program which converts file pathnames into inodes and print their information.

7.1 File Operation Levels

File operations consist of five levels, from low to high, as shown in the following hierarchy.

(1). **Hardware Level:** File operations at hardware level include

fdisk : divide a hard disk, USB or SDC drive into partitions.
mkfs : format disk partitions to make them ready for file systems.
fsck : check and repair file system.
defragmentation: compact files in a file system.

Most of these are system-oriented utility programs. An average user may never need them, but they are indispensable tools for creating and maintaining file systems.

K. C. Wang, *Systems Programming in Unix/Linux*, https://doi.org/10.1007/978-3-319-92429-8_7

(2). **File System Functions in OS Kernel:** Every operating system kernel provides support for basic file operations. The following lists some of these functions in a Unix-like system kernel, where the prefix k denotes kernel functions.

```
kmount(), kumount()           (mount/umount file systems)
kmkdir(), krmdir()            (make/remove directory)
kchdir(), kgetcwd()           (change directory, get CWD pathname)
klink(),  kunlink()           (hard link/unlink files)
kchmod(), kchown(), kutime()  (change r|w|x permissions,owner,time)
kcreat(), kopen()             (create/open file for R,W,RW,APPEND)
kread(),  kwrite()            (read/write opened files)
klseek(); kclose()            (lseek/close file descriptors)
ksymlink(), kreadlink()       (create/read symbolic link files)
kstat(),  kfstat(), klstat()  (get file status/information)
kopendir(), kreaddir()        (open/read directories)
```

(3). **System Calls:** User mode programs use system calls to access kernel functions. As an example, the following program reads the second 1024 bytes of a file.

```
#include <fcntl.h>
int main(int argc, char *argv[ ]) // run as a.out filename
{
   int fd, n;
   char buf[1024];
   if ((fd = open(argv[1], O_RDONLY)) < 0) // if open() fails
       exit(1);
   lseek(fd, 1024, SEEK_SET);     // lseek to byte 1024
   n = read(fd, buf, 1024);       // try to read 1024 bytes
   close(fd);
}
```

The functions **open()**, **read(), lseek()** and **close()** are C library functions. Each of these library functions issues a system call, which causes the process to enter kernel mode to execute a corresponding kernel function, e.g. open goes to kopen(), read goes to kread(), etc. When the process finishes executing the kernel function, it returns to user mode with the desired results. Switch between user mode and kernel mode requires a lot of actions (and time). Data transfer between kernel and user spaces is therefore quite expensive. Although it is permissible to issue a read(fd, buf, 1) system call to read only one byte of data, it is not very wise to do so since that one byte would come with a terrific cost. Every time we have to enter kernel, we should do as much as we can to make the journey worthwhile. In the case of read/write files, the best way is to match what the kernel does. The kernel reads/writes files by block size, which ranges from 1KB to 8KB. For instance, in Linux, the default block size is 4KB for hard disks and 1KB for floppy disks. So each read/write system call should also try to transfer one block of data at a time.

(4). **Library I/O Functions:** System calls allow the user to read/write chunks of data, which are just a sequence of bytes. They do not know, nor care, about the meaning of the data. A user often needs to read/write individual chars, lines or data structure records, etc. With only system calls, a user mode program must do these operations from/to a buffer area by itself. Most users would consider this too inconvenient. The C library provides a set of standard I/O functions for convenience, as well as for run-time efficiency. Library I/O functions include:

```
FILE mode I/O: fopen(),fread();  fwrite(),fseek(),fclose(),fflush()
char mode I/O: getc(), getchar() ugetc(); putc(),putchar()
line mode I/O: gets(), fgets();  puts(), fputs()
formatted I/O: scanf(),fscanf(),sscanf(); printf(),fprintf(),sprintf()
```

With the exceptions of sscanf()/sprintf(), which read/write memory locations, all other library I/O functions are built on top of system calls, i.e. they ultimately issue system calls for actual data transfer through the system kernel.

(5). **User Commands:** Instead of writing programs, users may use Unix/Linux commands to do file operations. Examples of user commands are

```
     mkdir, rmdir, cd, pwd, ls, link, unlink, rm, cat, cp, mv, chmod, etc.
```

Each user command is in fact an executable program (except cd), which typically calls library I/O functions, which in turn issue system calls to invoke the corresponding kernel functions. The processing sequence of a user command is either

```
   Command => Library I/O function => System call => Kernel Function
OR Command ======================== > System call => Kernel Function
```

(6). **Sh Scripts:** Although much more convenient than system calls, commands must be entered manually, or in the case of using GUI, by dragging file icons and clicking the pointing device, which is tedious and time-consuming. Sh scripts are programs written in the sh programming language, which can be executed by the command interpreter sh. The sh language include all valid Unix/Linux commands. It also supports variables and control statements, such as if, do, for, while, case, etc. In practice, sh scripts are used extensively in Unix/Linux systems programming. In addition to sh, many other script languages, such as Perl and Tcl, are also in wide use.

7.2 File I/O Operations

Figure 7.1 shows the diagram of file I/O operations.

In Fig. 7.1, the upper part above the double line represents kernel space and the lower part represents user space of a process. The diagram shows the sequence of actions when a process read/ write a file stream. Control flows are identified by the labels (1) to (10), which are explained below.

```
------------------------- User Mode Operations --------------------------------
```

(1). A process in User mode executes

```
  FILE *fp = fopen("file", "r"); or FILE *fp = fopen(“file”, “w”);
```

which opens a file stream for READ or WRITE.

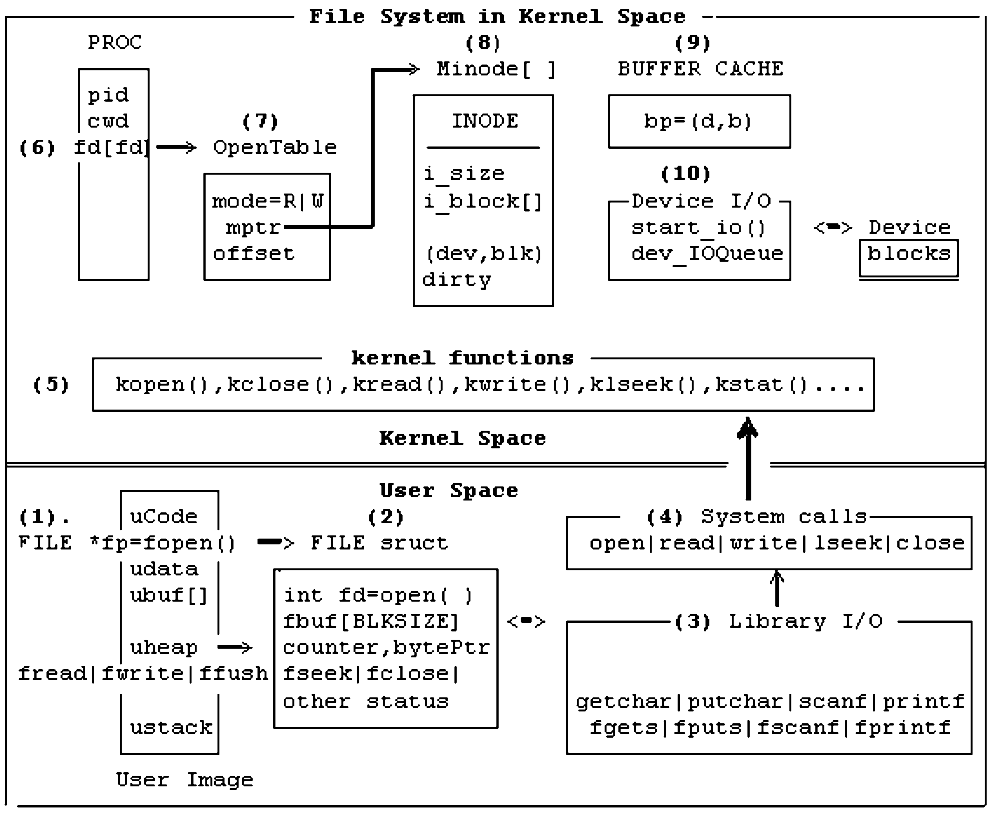


Fig. 7.1 File Operation Diagram

(2). fopen() creates a FILE structure in user (heap) space containing a file descriptor, fd, a fbuf [BLKSIZE] and some control variables. It issues a fd = open("file", flags=READ or WRITE) syscall to kopen() in kernel, which constructs an OpenTable to represent an instance of the opened file. The OpenTable's mptr points to the file's INODE in memory. For non-special files, the INODE's i_block array points to data blocks on the storage device. On success, fp points to the FILE structure, in which fd is the file descriptor returned by the open() syscall.

(3). fread(ubuf, size, nitem, fp): READ nitem of size each to ubuf by

```
. copy data from FILE structure's fbuf to ubuf, if enough, return;
. if fbuf has no more data, then execute (4a).
```

(4a). issue read(fd, fbuf, BLKSIZE) system call to read a file block from kernel to fbuf, then copy data to ubuf until enough or file has no more data.

(4b). fwrite(ubuf, size, nitem, fp): copy data from ubuf to fbuf;

```
. if (fbuf has room): copy data to fbuf, return;
. if (fbuf is full) : issue write(fd, fbuf, BLKSIZE) system call to write a
                      block to kernel, then write to fbuf again.
```

Thus, fread()/fwrite() issue read()/write() syscalls to kernel, but they do so only if necessary and they transfer data in chunks of block size for better efficiency. Similarly, other Library I/O Functions, such as fgetc/fputc, fgets/fputs, fscanf/fprintf, etc. also operate on fbuf in the FILE structure, which is in user space.

```
========================  Kernel Mode Operations   =======================
```

(5). File system functions in kernel:

```
Assume read(fd, fbuf[ ], BLKSIZE) system call of non-special file.
```

(6). In a read() system call, fd is an opened file descriptor, which is an index in the running PROC's fd array, which points to an OpenTable representing the opened file.

(7). The OpenTable contains the files's open mode, a pointer to the file's INODE in memory and the current byte offset into the file for read/write. From the OpenTable's offset,

```
  . Compute logical block number, lbk;
  . Convert logical block number to physical block number, blk, via INODE.i_block
[ ] array.
```

(8). Minode contains the in-memory INODE of the file. The INODE.i_block[ ] array contains pointers to physical disk blocks. A file system may use the physical block numbers to read/write data from/to the disk blocks directly, but these would incur too much physical disk I/O.

(9). In order to improve disk I/O efficiency, the OS kernel usually uses a set of I/O buffers as a cache memory to reduce the number of physical I/O. Disk I/O buffer management will be covered in Chap. 12.

(9a). For a read(fd, buf, BLKSIZE) system call, determine the needed (dev, blk) number, then consult the I/O buffer cache to

```
.get a buffer = (dev, blk);
.if (buffer's data are invalid){
    start_io on buffer;
    wait for I/O completion;
 }
.copy data from buffer to fbuf;
.release buffer to buffer cache;
```

(9b). For a write(fd, fbuf, BLKSIZE) system call, determine the needed (dev, blk) number, then consult the I/O buffer cache to

```
.get a buffer = (dev, blk);
.write data to the I/O buffer;
.mark buffer as dataValid and DIRTY (for delay-write to disk);
.release the buffer to buffer cache;
```

(10). Device I/O: Physical I/O on the I/O buffers ultimately go through the device driver, which consists of start_io() in the upper-half and disk interrupt handler in the lower-half of the driver.

```
------------------- Upper-half of disk driver --------------------
start_io(bp): //bp=a locked buffer in dev_list, opcode=R|W(ASYNC)
{
   enter bp into dev's I/O_queue;
   if (bp is FIRST in I/O_queue)
      issue I/O command to device;
}

------------------ Lower-half of disk driver --------------------
Device_Interrupt_Handler:
{
   bp = dequeue(first buffer from dev.I/O_queue);
   if (bp was READ){
       mark bp data VALID;
       wakeup/unblock waiting process on bp;
   }
   else       // bp was for delay write
       release bp into buffer cache;

   if (dev.I/O_queue NOT empty)
      issue I/O command for first buffer in dev.I/O_queue;
}
```

7.3 Low Level File Operations

7.3.1 Partitions

A block storage device, e.g. hard disk, USB drive, SD card, etc. can be divided into several logical units, called partitions. Each partition can be formatted as a specific file system and may be installed with a different operating system. Most booting programs, e.g. GRUB, LILO, etc. can be configured to boot up different operating systems from the various partitions. The partition table is at the byte offset 446 (0x1BE) in the very first sector, which is called the **Master Boot Record (MBR)** of the device. The table has 4 entries, each defined by a 16-byte partition structure, which is

```
 stuct partition {
       u8  drive;         // 0x80 - active
       u8  head;          // starting head
       u8  sector;        // starting sector
       u8  cylinder;      // starting cylinder
       u8  sys_type;      // partition type
       u8  end_head;      // end head
       u8  end_sector;    // end sector
       u8  end_cylinder;  // end cylinder
      u32  start_sector;  // starting sector counting from 0
      u32  nr_sectors;    // number of sectors in partition
};
```

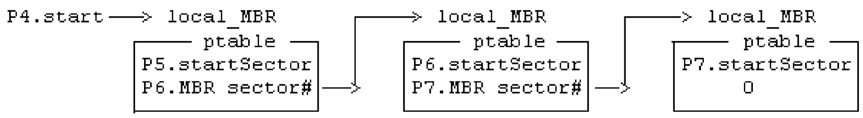


Fig. 7.2 Link List of Extend Partitions

If a partition is EXTEND type (type number =5), it can be divided into more partitions. Assume that partition P4 is EXTEND type and it is divided into extend partitions P5, P6, P7. The extend partitions form a link list in the extend partition area, as shown in Fig. 7.2.

The first sector of each extend partition is a **local MBR**. Each local MBR also has a partition table at the byte offset 0x1BE, which contains only two entries. The first entry defines the start sector and size of the extend partition. The second entry points to the next local MBR. All the local MBR's sector numbers are relative to P4's start sector. As usual, the link list ends with a 0 in the last local MBR. In a partition table, the CHS values are valid only for disks smaller than 8GB. For disks larger than 8GB but fewer than 4G sectors, only the last 2 entries, **start_sector** and **nr_sectors**, are meaningful. In the following, we demonstrate fdisk and partitions by examples. Since working with real disks of a computer is very risky, we shall use a virtual disk image, which is just an ordinary file but looks like a real disk.

(1). Under Linux, e.g. Ubuntu, create a virtual disk image file named mydisk.

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

dd is a program which writes 1440 (1KB) blocks of zeros to the target file mydisk. We choose count=1440 because it is the number of 1KB blocks of old floppy disks. If desired, the reader may specify a larger block number.

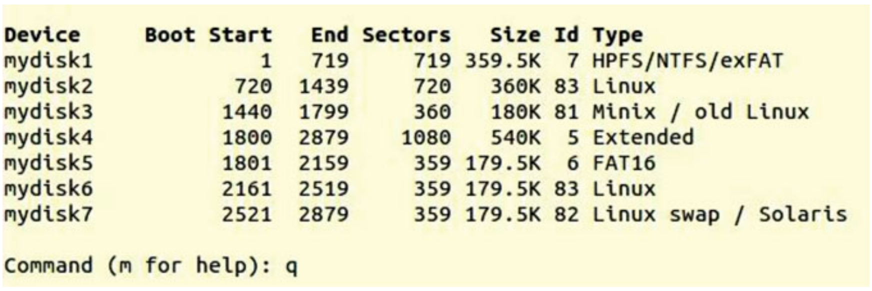
(2). Run fdisk on the disk image file:

```
fdisk mydisk
```

fdisk is an interactive program. It has a help menu showing all the commands. It collects inputs from the user to create a partition table in memory, which is written to the MBR of the disk image only if the user enters the w command. While in memory, it allows the user to create, examine and modify the partitions. After writing the partition to disk, exit fdisk by the q command. The following figure shows the result of a fdisk session, which partitions the disk image into 3 primary partitions (P1 to P3) and an extend partition P4. The extend partition P4 is further divided into more partitions (P5 to P7), all within the extend partition area. The partition types are all default to Linux when first created. They can be changed to different file system types by the t command. Each file system type has a unique value in HEX, e.g. 0x07 for HPFS/NTFS, 0x83 for Linux, etc. which can be displayed by the t command (Fig. 7.3).

Exercise 7.1: Write a C program to display the partitions, including extend partitions, if any, of a virtual disk image in the same format as does fdisk.

Helps: For readers with less programming experiences, the following may be of help.



```
Device     Boot Start   End Sectors   Size Id Type
mydisk1             1   719     719 359.5K  7 HPFS/NTFS/exFAT
mydisk2           720  1439     720   360K 83 Linux
mydisk3          1440  1799     360   180K 81 Minix / old Linux
mydisk4          1800  2879    1080   540K  5 Extended
mydisk5          1801  2159     359 179.5K  6 FAT16
mydisk6          2161  2519     359 179.5K 83 Linux
mydisk7          2521  2879     359 179.5K 82 Linux swap / Solaris

Command (m for help): q
```

Fig. 7.3. Partition Table

(1). Access partition tables in MBR:

```
#include <stdio.h>
#include <fcntl.h>
char buf[512];

int fd = open("mydisk", O_RDONLY);  // open mydisk for READ
read(fd, buf, 512);                 // read MBR into buf[512]
struct partition *p = (struct partition *)&buf[0x1BE];
printf("p->start_sector = %d\n", p->start_sector); // access p->start_sector
p++;        // advance p to point to the next partition table in MBR, etc.
```

(2). Assume that P4 is extend type (type=5) with start_sector = n.

```
lseek(fd, (long)(n*512), SEEK_SET);    // lseek to P4 begin
read(fd, char buf[ ], 512);            // read local MBR
p = (struct partition *)&buf[0x1BE]);  // access local ptable
```

(3). Extend partitions form a "link list", which ends with a NULL "pointer".

Conversely, we can also write a simple mkfs program to partition a virtual disk by the following steps.

(1). Open a virtual disk for read/write (O_RDWR).
(2). Read MBR into a char buf[512]; In a real disk, the MBR may contain the beginning part of a boot program, which must not be disturbed.
(3). Modify the partition table entries in buf[ ]: In each partition table entry, only the fields type, start_sector and nr_sectors are important. All other fields do not matter, so they can all be left as they are or set to zeros.
(4). Write buf[ ] back to the MBR.

Exercise 7.2: Write a C program myfdisk to divide a virtual disk into 4 primary partitions. After running the program, run Linux fdisk on the virtual disk to verify the results.

7.3.2 Format Partitions

fdisk merely divides a storage device into partitions. Each partition has an intended file system type, but the partition is not yet ready for use. In order to store files, a partition must be made ready for a specific file system first. The operation is traditionally called **format** a disk or disk partition**.** In Linux, it is called **mkfs**, which stands for Make File System. Linux supports many different kinds of file systems. Each file system expects a specific format on the storage device. In Linux, the command

```
mkfs -t TYPE [-b bsize] device nblocks
```

makes a file system of TYPE on a device of nblocks, each block of bsize bytes. If bsize is not specified, the default block size is 1KB. To be more specific, we shall assume the EXT2/3 file system, which used to be the default file systems of Linux. Thus,

```
mkfs -t ext2 vdisk 1440   OR  mke2fs vdisk 1440
```

formats vdisk to an EXT2 file system with 1440 (1KB) blocks. A formatted disk should be an empty file system containing only the root directory. However, mkfs of Linux always creates a default lost +found directory under the root directory. After mkfs, the device is ready for use. In Linux, a new file system is not yet accessible. It must be mounted to an existing directory in the root file system. The / mnt directory is usually used to mount other file systems. Since virtual file systems are not real devices, they must be mounted as loop devices, as in

```
sudo mount -o loop vdisk /mnt
```

which mounts vdisk onto the /mnt directory. The mount command without any parameter shows all the mounted devices of the Linux system. After mounting, the mounting point /mnt becomes equivalent to the root directory of the mounted device. The user may change directory (cd) into /mnt, do file manipulations on the device as usual. After using the mounted device, cd out of /mnt, then enter

```
sudo umount /mnt   OR   sudo umount vdisk
```

to un-mount the device, detaching it from the root file system. Files saved on the device should be retained in that device.

It is noted that most current Linux systems, e.g. Slackware 14.1 and Ubuntu 15.10, can detect portable devices containing file systems and mount them directly. For example, when a Ubuntu user inserts a USB drive into a USB port, Ubuntu can detect the device, mount it automatically and display the files in a pop up window, allowing the user to access files directly. The user may enter the **mount** command to see where the device (usually /dev/sdb1) is mounted on, as well as the mounted file system type. If the user pulls out the USB drive, Ubuntu will detect it also and **umount** the device automatically. However, pulling out a portable device arbitrarily may corrupt data on the device. This is because the Linux kernel typically uses delayed writes of data to devices. To ensure data consistency, the user should umount the device first before detaching it.

The Linux mount command can mount partitions of real devices or an entire virtual disk, but it can not mount partitions of a virtual disk. If a virtual disk contains partitions, the partitions must be associated with **loop devices** first. The following example demonstrates how to mount partitions of virtual disks.

7.3.3 Mount Partitions

Man 8 losetup: display losetup utility command for system administration

(1). Create a virtual disk image by the dd command:

```
dd if=/dev/zero of=vdisk bs=1024 count=32768   #32K (1KB) blocks
```

(2). Run fdisk on vdisk to create a partition P1

```
fdisk vdisk
```

Enter the n (new) command to create a partition P1 by using default start and last sector numbers. Then, enter the w command to write the partition table to vdisk and exit fdisk. vdisk should contain a partition P1=[start=2048, end=65535]. The partition is 63488 sectors.

(3). Create a loop device on partition 1 of vdisk using sector numbers

```
losetup -o $(expr 2048 \* 512) --sizelimit $(expr 65535 \* 512) /dev/loop1
vdisk
```

losetup needs the start byte (start_sector*512) and end byte (end_sector*512) of the partition. The reader may compute these values by hand and use them in the losetup command. Loop devices for other partitions can be set up in a similar way. After creating a loop device, the reader may use the command

```
losetup -a
```

which displays all loop devices as /dev/loopN.

(4). Format /dev/loop1 an EXT2 file system

```
mke2fs -b 4096 /dev/loop1 7936           # mke2fs with 7936 4KB blocks
```

The size of the partition is 63488 sectors. The number of 4KB blocks is 63488 / 8 = 7936

(5). Mount the loop device

```
mount /dev/loop1 /mnt                    # mount as loop device
```

(6). Access mounted device as part of the file system

```
(cd /mnt; mkdir bin boot dev etc user)   # populate with DIRs
```

(7). When finished with the device, umount it.

```
umount /mnt
```

(8). When finished with a loop device, detach it by the command

```
losetup -d /dev/loop1                    # detach a loop device.
```

Exercise 7.3: Partition a vdisk into 4 partitions. Create 4 loop devices for the partitions. Format the partitions as EXT2 file systems, mount the partitions and copy files into them. Then umount the loop devices and detach the loop devices. If the reader knows how to write sh scripts, all the above steps can be done by a simple sh script.

7.4 Introduction to EXT2 File System

For many years, Linux used EXT2 (Card et al. 1995; EXT2 2001) as the default file system. EXT3 (EXT3 2015) 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.

7.4.1 EXT2 File System Data Structures

Under Linux, we can create a virtual disk containing a simple EXT2 file system as follows.

```
(1). dd if=/dev/zero of=mydisk bs=1024 count=1440
(2). mke2fs -b 1024 mydisk 1440
```

The resulting EXT2 file system has 1440 blocks, each of block size 1KB bytes. We choose 1440 blocks because it is the number of blocks of (old) floppy disks. The resulting disk image can be used directly as a virtual (floppy) disk on most virtual machines that emulate the Intel x86 based PCs, e.g. QUEM, VirtualBox and Vmware, etc. The layout of such an EXT2 file system is shown in Figure 7.4.

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

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

7.4.2 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.

```
| 0  |  1  |  2 |  3-7    | 8  | 9  | 10 | . . . 32|33          1439|
-------------------------------------------------------------------
|boot|super| GD |reserved|bmap|imap|inodes blocks |   data blocks   |
-------------------------------------------------------------------
```

Fig. 7.4 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 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_mount_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**.

7.4.3 Group Descriptor

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

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

Since a FD has only 1440 blocks, B2 contains only 1 group descriptor. The rest are 0's. On 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.

7.4.4 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.

7.4.5 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. For example, 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 hour, 0 minute, 0 second 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 (1KB or 4KB) 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.

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 /.

7.4.6 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 byte. So the dir_entry's rec_len also varies.

7.5 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
```

7.5.1 Display Superblock

The following C program in Example 7.1 displays the superblock of an EXT2 file system. The basic technique is to read the superblock (block #1 or 1KB at offset 1024) into a char buf[1024]. Let a struct ext2_super_block *p point to buf[ ]. Then use p->field to access the various fields of the superblock structure. The technique is similar to that of accessing partition tables in the MBR.

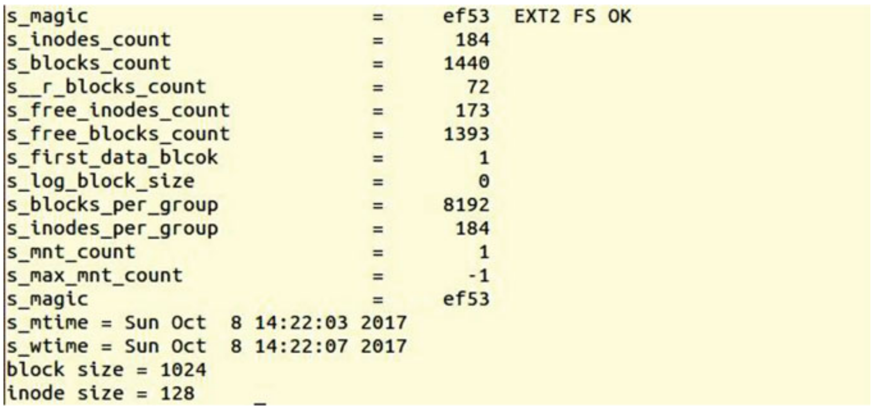
Example 7.1: superblock.c program: display superblock information of an EXT2 file system.

```
/*********** 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 or offset 1024
  read(fd, buf, 1024);
  sp = (SUPER *)buf;           // as a super block structure
  // check for 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_blcok",  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 7.5 shows the outputs of running the super.c example program.

Exercise 7.4: Write a C program to display the group descriptor of an EXT2 file system on a device.



```
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. 7.5 Superblock of Ext2 File System

7.5.2 Display Bitmaps

The C program in Example 7.2 display the inodes bitmap (imap) in HEX form.

Example 7.2: 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, 0);
  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;
  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 ", (unsigned char)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 are printed in reverse order, i.e. from high to low address.

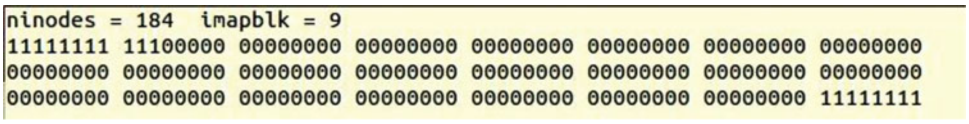
Exercise 7.5: Modify the imap.c program to print the inodes bitmap in char map form, i.e. for each bit, print a '0' if the bit is 0, print a '1' if the bit is 1.

Hints:
(1). Examine the bit patterns of (1 << j) for j=0 to 7.
(2). If char c is a byte, the C statement

```
                  if ( c & (1<<j) )    (j=0 to 7)
```

tests whether bit j of c is 1 or 0.

The outputs of the modified program should look like Fig. 7.6, in which each char represents a bit in the imap.



```
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. 7.6. Inodes Bitmap of an EXT2 File System

Exercise 7.6: Write a C program to display the blocks bitmap of an Ext2 file system, also in char map form.

7.5.3 Display root Inode

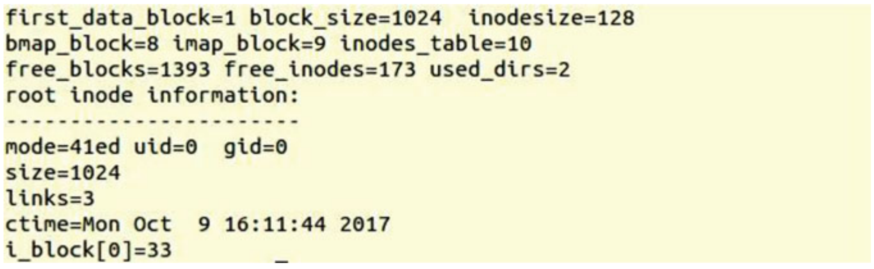
In an EXT2 file system, the number 2 (count from 1) inode is the inode of the root directory /. If we read the root inode into memory, we should be able to display its various fields such as mode, uid, gid, file size, creation time, hard link count and data block numbers, etc. The program in Example 7.3 displays the INODE information of the root directory of an EXT2 file system.

Example 7.3: 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"; // default to 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) < 0);
  if (fd < 0){
    printf("open failed\n"); exit(1);
  }
  get_block(fd, 1, buf);               // get superblock
  sp = (SUPER *)buf;
  firstdata = sp->s_first_data_block;
  inodesize = sp->s_inode_size;
  blksize = 1024*(1<<sp->s_log_block_size);
  printf("first_data_block=%d block_size=%d inodesize=%d\n",
          firstdata, blksize, inodesize);
  get_block(fd, (firstdata+1), 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,
          gp->bg_free_blocks_count);
  printf("free_blocks=%d free_inodes=%d\n",
          gp->bg_free_inodes_count,
          gp->bg_used_dirs_count);
  iblock = gp->bg_inode_table;
  printf("root inode information:\n", iblock);
  printf("----------------------\n");
  get_block(fd, iblock, buf);
  ip = (INODE *)buf + 1;             // 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 7.7 shows the root inode of an EXT2 file system.

In Fig. 7.7, the 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 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.



```
first_data_block=1 block_size=1024  inodesize=128
bmap_block=8 imap_block=9 inodes_table=10
free_blocks=1393 free_inodes=173 used_dirs=2
root inode information:
-----------------------
mode=41ed uid=0  gid=0
size=1024
links=3
ctime=Mon Oct  9 16:11:44 2017
i_block[0]=33
```

Fig. 7.7 root Inode of EXT2 file system

7.5.4 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
  };
```

Thus, the contents of each data block of a directory has the form

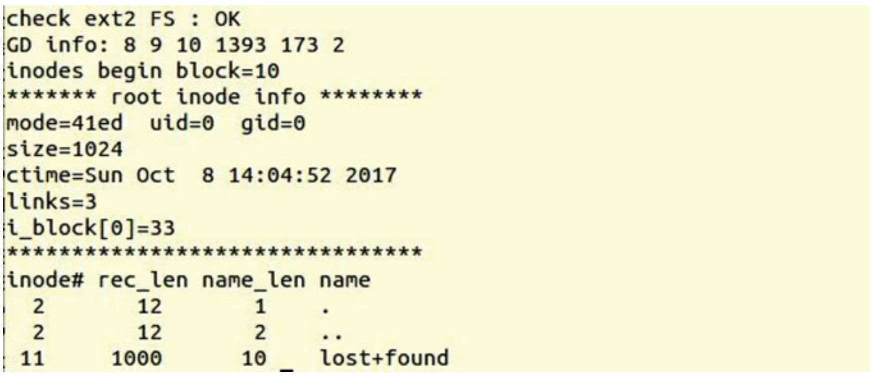
```
         [inode rec_len name_len NAME] [inode rec_len name_len NAME] ......
```

where NAME is a sequence of name_len chars (without a terminating NULL). 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;                 // in temp[ ]
  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
}
```

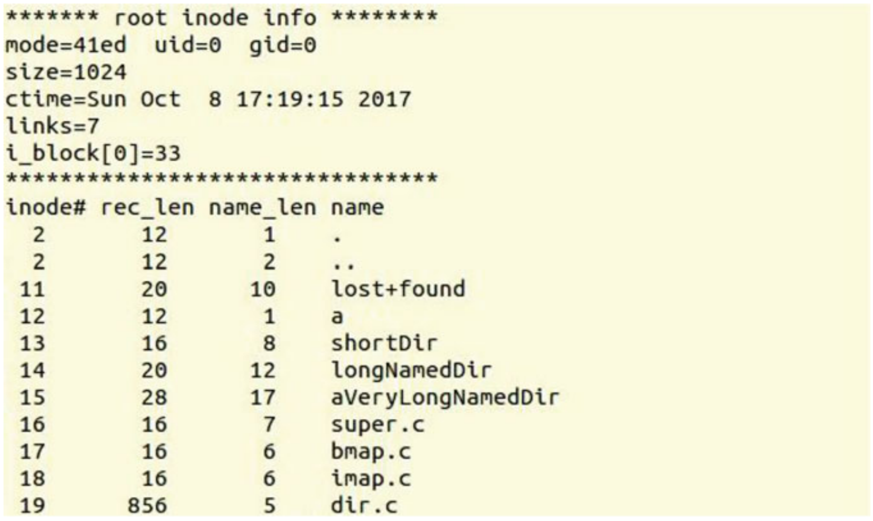
Exercise 7.7: Write a C program to print the dir_entries of a directory. For simplicity, we may assume a directory INODE has at most 12 direct blocks, i_block[0] to i_block[11]. This assumption is reasonable. With 1KB block size and an average file name length of 16 chars, a single disk block can contain 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. So we only need to go through at most 12 direct blocks. For an empty EXT2 file system, the program's output should look like Fig. 7.8.

Exercise 7.8: Mount mydisk under Linux. Create new directories and copy files to it. Umount it. Then run the dir.c program on mydisk again to see the outputs, which should look like Fig. 7.9. The reader should have noticed that the name_len of each entry is the exact number of chars in the name field, and



```
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. 7.8 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. 7.9 List Entries of a Directory

every rec_len is a multiple of 4 (for memory 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 7.9: Assume INODE is the struct ext2_inode type. Given an inode number, ino, write C statements to return a pointer to its INODE structure.

Hints:
(1). int fd = Open vidsk for READ;
(2). int offset = inodes_start_block * BLKSIZE + (ino-1)*inode_szie;
(3). lseek fd to offset;
(4). read inode_size bytes into an INODE inode;
(5). return (INODE *ip = &inode);

Exercise 7.10: Given an INODE pointer to a DIRectory inode. Write a

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

function which searches for a name string in the directory; return its inode number if found, return 0 if not.

7.6 Programming Project: Convert File Pathname to Inode

In every file system, almost every operation starts with a filename, e.g.

```
cat filename
open filename for R|W
copy file1 to file2
etc.
```

The fundamental problem of every file system is to convert a filename into the file's representation data structure in the file system. In the case of EXT2/3/4 file systems, it amounts to converting a pathname to the file's INODE. The programming project is to integrate the knowledge and programming techniques in the above examples and exercises into a program which finds a file in an EXT2 file system and prints its information. Assume vdisk is a virtual disk containing many levels of directories and files. Given the pathname of a file, e.g. /a/b/c/d, write a C program to find the file and prints its information, such as the file type, owner id, size, date of creation and data block numbers, including indirect and double indirect blocks.

HINTS and helps:

(1). Tokenize pathname into component strings. Denote them as char *name[0], *name[1], .. ,*name[n-1], where n is the number of component strings.

(2). Start from INODE *ip -> root inode (ino=2)

(3). for (int i=0; i<n; i++){

```
    int ino = search(ip, name[i]);
    if (!ino) exit(1);              // can't find name[i]
    ip ->inode of name[i];          // ip point at INODE of name[i]
    if (*ip is not a DIR) exit(2);  // name[i] is not a DIR
}
```

(4). If reach here: ip must point at file's inode. Use ip-> to access the fields of inode

(5). To print indirect block numbers:

```
read i_block[12] into a char buf[BLKSIZE];
u32 *up = (u32 *)buf;
*up is an indirect block number; up++ points to the next number, etc.
```

Similar technique can be used to print double indirect block numbers.

7.7 Summary

This chapter covers file systems. It explains the various levels of file operations in operating systems. These include preparing storage devices for file storage, file system support functions in kernel, system calls, library I/O functions on file streams, user commands and sh scripts for file operations. It presents a systematic overview of file operations, from read/write file streams in user space to system calls to kernel space down to the device I/O driver level. It describes low level file operations, which include disk partitions, example programs to display partition tables, format partitions for file systems and mount disk partitions. It presents an introduction to the EXT2 file system of Linux, which include the EXT2 file system data structures, program code to display superblock, group descriptor, blocks and inodes bitmaps and directory contents. The programming project is to integrate the knowledge and programming techniques presented in the chapter into a program which finds a file in an EXT2 file system and prints its information. The purpose of the project is to let the reader understand how to convert a file's pathname into its inode, which is the fundamental problem of any file system.

Problems

1. Modify the program of Exercise 7.2 to support extend partitions.
2. Large EXT2/3 file system comprise more than 32K blocks of 4KB block sizes. Extend the Project program to large EXT2/3 file systems.

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

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