0107451 Ext2 file system

Department of Communications and Computer Engineering
Tafila Technical University

This is a two-week programming project assignment.

1 How to create an Ext2 file system

To create an Ext2 file system on a floppy disk we will use the *fdformat* and *mkfs.ext2* commands. fdformat is used to format the disk, while *mkfs.ext2* creates the file system. The commands work as following:

```
[ealtieri@italia os]$ fdformat /dev/fd0
Double-sided, 80 tracks, 18 sec/track. Total capacity 1440 kB.
Formatting ... done
Verifying ... done

[ealtieri@italia os]$ /sbin/mkfs.ext2 /dev/fd0
mke2fs 1.26 (3-Feb-2002)
Filesystem label=
OS type: Linux
Block size=1024 (log=0)
Fragment size=1024 (log=0)
184 inodes, 1440 blocks
72 blocks (5.00%) reserved for the super user
First data block=1
1 block group
8192 blocks per group, 8192 fragments per group
184 inodes per group

Writing inode tables: done
Writing superblocks and filesystem accounting information: done

This filesystem will be automatically checked every 33 mounts or
180 days, whichever comes first.  Use tune2fs -c or -i to override.
```

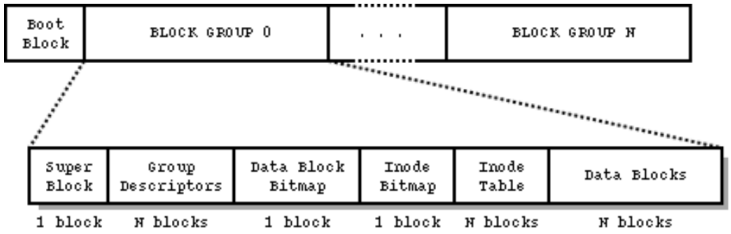
In the example above, */dev/fd0* is the device in charge of raw access to 1.44 MB floppy disks. We will use the same device in our examples later.
Once the two commands terminate, you may test the floppy:

```
[ealtieri@italia os]$ mount /mnt/floppy/
[ealtieri@italia os]$ ls -sil /mnt/floppy/
total 12
11   12 drwx------    2 root     root        12288 Jun 24 12:14 lost+found
[ealtieri@italia os]$ umount /mnt/floppy/
```

Notice the *-sil* options in the *ls* command. The meaning of these options will be explained later.

2 Structure of an Ext2 File system

On disk, the Ext2 filesystem is organized as shown in the picture below:



The first 1024 bytes of the disk, the *"boot block"*, are reserved for the partition boot sectors and are unused by the Ext2 filesystem. The rest of the partition is split into block groups, each of which has the layout shown in the figure above. On a 1.44 MB floppy disk, there is only one block group.

The Ext2 data structures above are defined in *include/linux/ext2_fs.h*. In the following sections we will go through each of these data structures.

3 The SuperBlock

The superblock is defined in *struct ext2_super_block*, line 339 of *include/linux/ext2_fs.h*. It contains information such as the total number of blocks on disk, the size of a block (usually 1024 bytes), the number of free blocks, etc. The meaning of each field in the *ext2_super_block structure* is explained in *ext2_fs.h*. Part of this structure has been reported below:

```
struct ext2_super_block {
    __u32 s_inodes_count; /* Inodes count */
    __u32 s_blocks_count; /* 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_blocks_per_group; /* # Blocks per group */
    ...
    __u16 s_magic; /* Magic signature */
    ...
```

The __u32, __u16 and __u8 data types denote unsigned 32-, 16- and 8-bit integer numbers.

s_inodes_count and *s_blocks_count* store the number of inodes and blocks on disk. If you look back at the output of *mkfs.ext2*, you will see that the total number of blocks on a floppy disk is 1440 and the number of inodes is 184. Also notice from the same output that the size of a block is 1024 bytes, therefore 1440 blocks * 1024 bytes = 1440 KB, the total size of a floppy disk.

The size of a block is given by *s_log_block_size*. This value expresses the size of a block as a power of 2, using 1024 bytes as the unit. Thus, 0 denotes 1024-byte blocks, 1 denotes 2048-byte blocks, and so on. To calculate the size in bytes of a block:

```
unsigned int block_size = 1024 << super.s_log_block_size;   /* block size in bytes */
```

The super block also tells us the number of blocks per group with *s_blocks_per_group*. Using *mkfs.ext2*, we can see that this value is 8192 on a floppy disk. Because there are 1440 blocks on a floppy, there can only be one group.

question Q1. What is the difference in terms of internal fragmentation and average file size between using a small block size (1024) and a big block size (e.g. 8192)? question Q2. What is the size in bytes of a block if *s_log_block_size=3* in the super-block?

The superblock is located at offset 1024 of a floppy. The code to read the superblock from a floppy is shown below. This code also checks the magic number of the super block (EXT2_SUPER_MAGIC) to see if we are reading from an Ext2 filesystem. For simplicity, error checking has been omitted.

```
#include <linux/ext2_fs.h>

...

int main()
{
    int fd;
    struct ext2_super_block super;

    fd = open("/dev/fd0", O_RDONLY);    /* open floppy device */

    lseek(fd, 1024, SEEK_SET);                    /* position head above super-block */
    read(fd, &super, sizeof(super));              /* read super-block */

    if (super.s_magic != EXT2_SUPER_MAGIC)
        exit(1); /* bad file system */

    block_size = 1024 << super.s_log_block_size;  /* calculate block size in bytes */

    ...
```

ext2super.c reads the superblock from a floppy disk and prints it to screen.

4 Group Descriptors

In the blocks immediately following the super-block reside the list of block-group descriptors. This list contains a descriptor for each block group on the disk. In the case of a floppy, there is only one block group and therefore one group descriptor. For a bigger disk, we would have to calculate the size of this list by using *s_blocks_count* and *s_blocks_per_group* in the superblock:

```
/* calculate number of block groups on the disk */
unsigned int group_count = 1 + (super.s_blocks_count-1) / super.s_blocks_per_group;

/* calculate size of the group descriptor list in bytes */
unsigned int descr_list_size = group_count * sizeof(struct ext2_group_descr);
```

A group descriptor is defined by the *ext2_group_descr* structure, line 148 of *ext2_fs.h*. This structure is reported below:

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

A 1.44 MB floppy has one group descriptor only, which can be read using the following code:

```
struct ext2_group_descr group_descr;

...
```

```
lseek(fd, 1024 + block_size, SEEK_SET);  /* position head above the group descriptor block */
read(fd, &group_descr, sizeof(group_descr));
```

ext2group.c reads the first (and only) group descriptor of a floppy disk and prints it to screen.

The group descriptor tells us the location of the block/inode bitmaps and of the inode table (described later) through the bg_block_bitmap, bg_inode_bitmap and bg_inode_table fields. These values indicate the blocks where the bitmaps and the table are located. It is handy to have a function to convert a block number to an offset on disk, which can be easily done by knowing that all blocks on disk have the same size of block_size bytes (calculated earlier from the super-block):

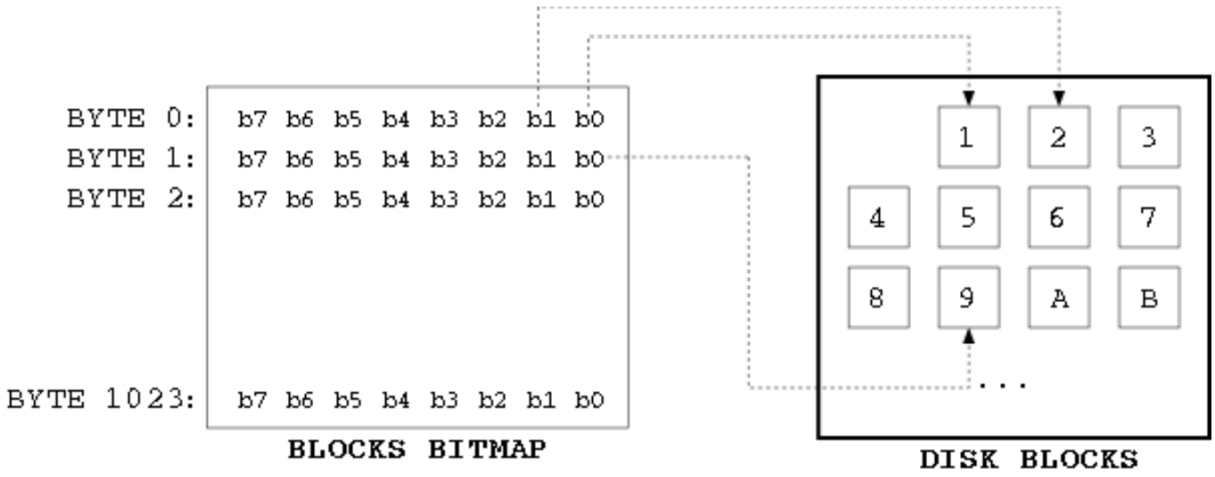
```
#define BASE_OFFSET 1024  /* location of the super-block in the first group */

#define BLOCK_OFFSET(block) (BASE_OFFSET + (block-1)*block_size)
```

Blocks are numbered starting from 1. Block 1 is the superblock of the first group, block 2 contains the group descriptors, and so on. Block 0 is the NULL block and does not correspond to any disk offset.

5 The blocks and inodes bitmaps

A bitmap is a sequence of bits. Each bit represents a specific block (blocks bitmap) or inode (inode bitmap) in the block group. A bit value of 0 indicates that the block/inode is free, while a value of 1 indicates that the block/inode is being used. A bitmap always refers to the block-group it belongs to, and its size must fit in one block.



Limiting the size of a bitmap to one block also limits the size of a block-group, because a bitmap always refers to the blocks/inodes in the group it belongs to. Consider the blocks bitmap: given a block size of 1024 bytes, and knowing that each byte is made of 8 bits, we can calculate the maximum number of blocks that the blocks bitmap can represent: 8 * 1024 = 8192 blocks. Therefore, 8192 blocks is the size of a block-group using a 1024-byte block size, as we also see from the output of mkfs.ext2 in the first section.

question Q3. Calculate the number of blocks in a block-group given a block size of 4096 bytes. The following code fragment reads the block bitmap from disk:

```
struct ext2_super_block super;  /* the super block */
struct ext2_group_desc group;   /* the group descritopr */
unsigned char *bitmap;

/* ... [read superblock and group descriptor] ... */
```

```
bitmap = malloc(block_size);    /* allocate memory for the bitmap */
lseek(fd, BLOCK_OFFSET(group->bg_block_bitmap), SEEK_SET);
read(fd, bitmap, block_size);   /* read bitmap from disk */

...

free(bitmap);
```

6 The inode table

The inode table consists of a series of consecutive blocks, each of which contains a predefined number of inodes. The block number of the first block of the inode table is stored in the *bg_inode_table* field of the group descriptor. To figure out how many blocks are occupied by the inode table, divide the total number of inodes in a group (stored in the *s_inodes_per_group* field of the superblock) by the number of inodes per block:

```
/* number of inodes per block */
unsigned int inodes_per_block = block_size / sizeof(struct ext2_inode);

/* size in blocks of the inode table */
unsigned int itable_blocks = super.s_inodes_per_group / inodes_per_block;
```

In the case of our floppy disk, we can see from the output of *mkfs.ext2* that we have 184 inodes per group and a block size of 1024 bytes. The size of an inode is 128 bytes, therefore the inode table will take 184 / (1024/128) = 23 blocks.

question Q4. Assume a block size of 2048 bytes. What is the offset of the inode table if *bg_inode_table=5* in the group descritor?

The inode table contains everything the operating system needs to know about a file, including the type of file, permissions, owner, and, most important, where its data blocks are located on disk. It is no surprise therefore that this table needs to be accessed very frequently and its read access time should be minimized as much as possible. Reading an inode from disk every time it is needed is usually a very bad idea. However, in this context we will adopt this method to keep the example code as simple as possible. We provide a general function to read an inode from the inode table:

```
static void read_inode(fd, inode_no, group, inode)
int                            fd;        /* the floppy disk file descriptor */
int                            inode_no;  /* the inode number to read  */
const struct ext2_group_desc *group;      /* the block group to which the inode belongs */
struct ext2_inode            *inode;      /* where to put the inode */
{
    lseek(fd, BLOCK_OFFSET(group->bg_inode_table)+(inode_no-1)*sizeof(struct ext2_inode), SEE
    read(fd, inode, sizeof(struct ext2_inode));
}
```

The offset of the inode to read is calculated by adding together the absolute offset of the inode table and the distance of the desired inode from the beginning of the inode table.

question Q5. Calculate the offset of inode 93 on disk. Assume a block size of 1024 bytes and *bg_inode_table=5*. To which block does the inode belong to? (The size of an inode structure is 128 bytes)

Inodes are numbered starting from 1. An inode is defined as a *struct ext2_inode*, in *ext2_fs.h*. The most important fields of this structure have been reported below:

```
struct ext2_inode {
    __u16   i_mode;         /* File type and access rights */
```

```
    __u16   i_uid;          /* Low 16 bits of Owner Uid */
    __u32   i_size;         /* Size in bytes */
    __u32   i_atime;        /* Access time */
    __u32   i_ctime;        /* Creation time */
    __u32   i_mtime;        /* Modification time */
    __u32   i_dtime;        /* Deletion Time */
    __u16   i_gid;          /* Low 16 bits of Group Id */
    __u16   i_links_count;  /* Links count */
    __u32   i_blocks;       /* Blocks count */
    __u32   i_flags;        /* File flags */
    ...
    __u32   i_block[EXT2_N_BLOCKS];  /* Pointers to blocks */
    ...
};
```

Most of the fields in this structure are self-explanatory. Particular atttention should be paid to the fields in bold, explained in the following paragraphs.

i_mode determines the type and access rights of a file. Possible file types are listed below. For each file type is defined a macro (sys/stat.h) that can be used to test for that specific file type.

```
Type                Macro
Regular file        S_ISREG(m)
Directory           S_ISDIR(m)
Character Device    S_ISCHR(m)
Block Device        S_ISBLK(m)
Fifo                S_ISIFO(m)
Socket              S_ISSOCK(m)
Symbolic Link       S_ISLNK(m)
```

The file permissions are also stored in *i_mode*. These permissions can be tested by ANDing *i_mode* with a set of symbols defined in sys/stat.h:

```
Domain  Read      Write      Exec       All
User    S_IRUSR   S_IWUSR    S_IXUSR    S_IRWXU
Group   S_IRGRP   S_IWGRP    S_IXGRP    S_IRWXG
All     S_IROTH   S_IWOTH    S_IXOTH    S_IRWXO
```

For example, to test if a file has user-execute permissions:

```
if (inode.i_mode & S_IXUSR) ...
```

The *i_blocks* field of the inode structure counts the number of blocks used by the file. Pointers to the actual data blocks of the file are stored in the *i_block[EXT2_N_BLOCKS]* array. The *EXT2_N_BLOCKS* symbol is defined in *ext2_fs.h* (line 177) as following:

```
#define EXT2_NDIR_BLOCKS 12                       /* number of direct blocks */
#define EXT2_IND_BLOCK EXT2_NDIR_BLOCKS         /* single indirect block   */
#define EXT2_DIND_BLOCK (EXT2_IND_BLOCK + 1)    /* double indirect block   */
#define EXT2_TIND_BLOCK (EXT2_DIND_BLOCK + 1)   /* trible indirect block   */
#define EXT2_N_BLOCKS (EXT2_TIND_BLOCK + 1)   /* total number of blocks  */
```

In total there are 15 pointers in the *i_block[]* array. The meaning of each of these pointers is explained below:

```
i_block[0..11] point directly to the first 12 data blocks of the file.
i_block[12] points to a single indirect block
i_block[13] points to a double indirect block
i_block[14] points to a triple indirect block
```

question Q6. Given a block size of 4096 bytes, what is the maximum size (in blocks and bytes) of a file without indirection? What is the total maximum size of a file? (Remember that each "block pointer" is declared as *__u32*, that is 4 bytes)
The following example prints the blocks used by the root directory of the floppy disk, which is always on inode 2 of the inode table:

```
struct ext2_inode inode;
struct ext2_group_desc group;

/* ... [ read superblock and group descriptor ] ... */

read_inode(fd, 2, &group, &inode);   /* read root inode (#2) from the floppy disk */

for(i=0; i<EXT2_N_BLOCKS; i++)
    if (i < EXT2_NDIR_BLOCKS)                                   /* direct blocks */
        printf("Block %2u : %u\n", i, inode.i_block[i]);
    else if (i == EXT2_IND_BLOCK)                               /* single indirect block */
        printf("Single   : %u\n", inode.i_block[i]);
    else if (i == EXT2_DIND_BLOCK)                              /* double indirect block */
        printf("Double   : %u\n", inode.i_block[i]);
    else if (i == EXT2_TIND_BLOCK)                              /* triple indirect block */
        printf("Triple   : %u\n", inode.i_block[i]);
```

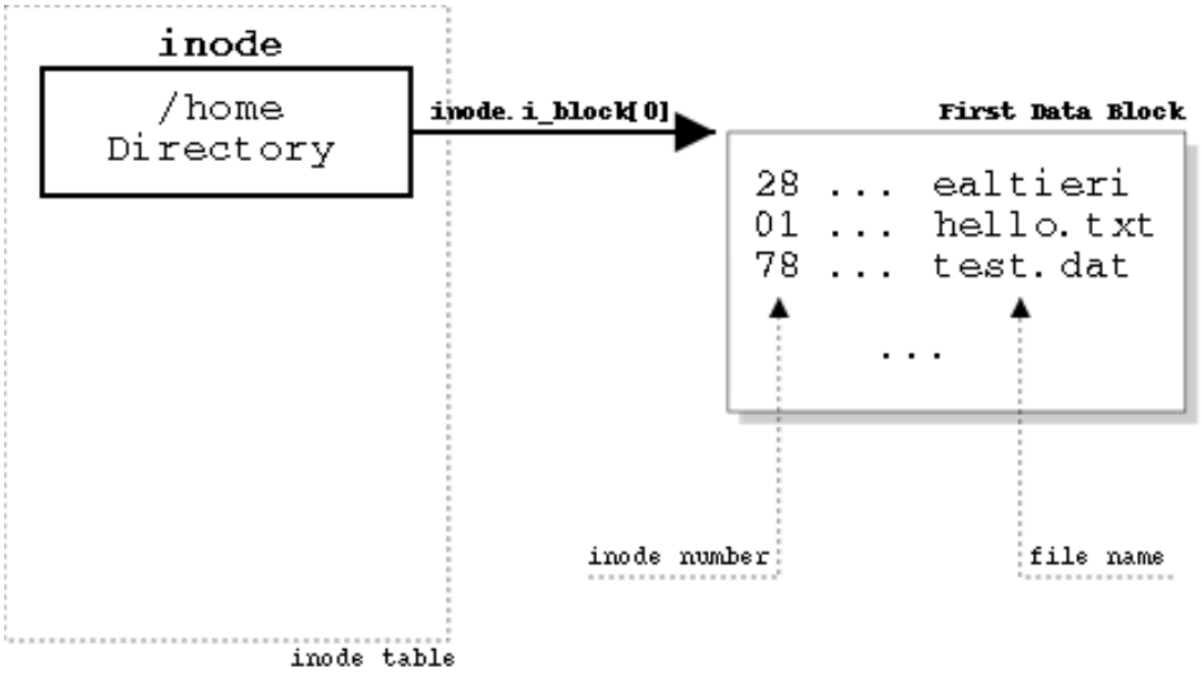
ext2root.c reads the root inode (#2) from the inode table of a floppy disk.

7 Directory entries in the inode table

Directory entries in the inode table require special attention. To test if an inode refers to a directory file we can use the *S_ISDIR(mode)* macro:

```
if (S_ISDIR(inode.i_mode)) ...
```

In the case of directory entries, the data blocks pointed by *i_block[]* contain a list of the files in the directory and their respective inode numbers.



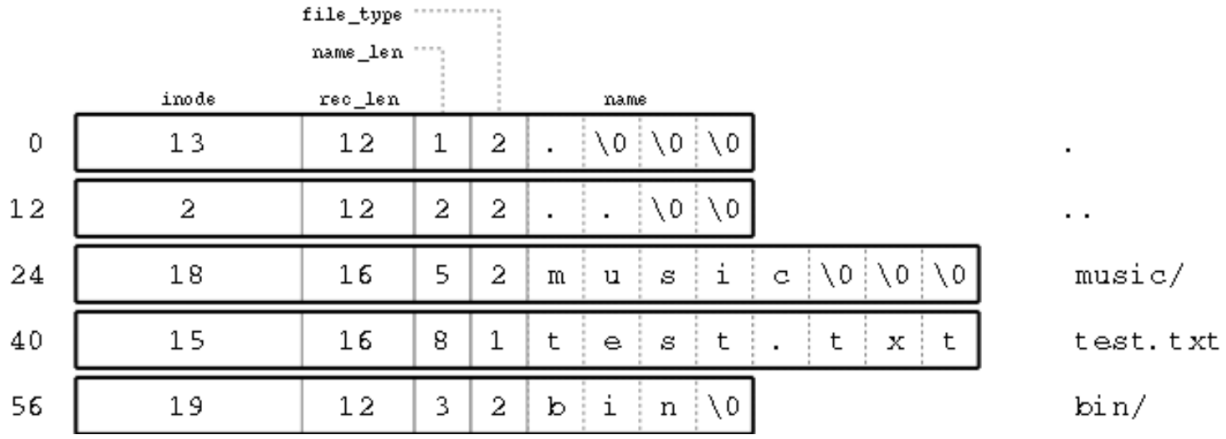
The list is composed of *ext2_dir_entry_2* structures:

```
struct ext2_dir_entry_2 {
    __u32 inode; /* Inode number */
    __u16 rec_len; /* Directory entry length */
    __u8 name_len; /* Name length */
    __u8 file_type;
    char name[EXT2_NAME_LEN]; /* File name */
};
```

The *file_type* field indicates what kind of file the entry is referring to. Possible values are:

file_type	Description
0	Unknown
1	Regular File
2	Directory
3	Character Device
4	Block Device
5	Named pipe
6	Socket
7	Symbolic Link

Each entry has a variable size depending on the length of the file name. The maximum length of a file name is *EXT2_NAME_LEN*, which is usually 255. The *name_len* field stores the length of the file name, while *rec_len* stores the total size of the entry and is used to locate the next entry in the list.



question Q7. Can you explain when and why NULL characters ("\0") are appended to the end of the file name? The following code reads the entries of a directory. Assume that the inode of the directory is stored in inode:

```
struct ext2_dir_entry_2 *entry;
unsigned int size;
unsigned char block[BLOCK_SIZE];

...

lseek(fd, BLOCK_OFFSET(inode->i_block[0]), SEEK_SET);
read(fd, block, block_size);                          /* read block from disk*/

size = 0;                                             /* keep track of the bytes read */
entry = (struct ext2_dir_entry_2 *) block;            /* first entry in the directory */
```

```
while(size < inode->i_size) {
    char file_name[EXT2_NAME_LEN+1];
    memcpy(file_name, entry->name, entry->name_len);
    file_name[entry->name_len] = 0;              /* append null char to the file name */
    printf("%10u %s\n", entry->inode, file_name);
    entry = (void*) entry + entry->rec_len;      /* move to the next entry */
    size += entry->rec_len;
}
```

The code above reads the first data block of a directory from disk. The block is stored in the block array. As explained earlier, this block contains a list of the directory's entries. The problem is that entries in this list have a variable size. This is the reason why we cannot just read the data block into an array of *struct ext2_dir_entry_2* elements.

The entry pointer points to the current entry in the directory. The file name of the entry is copied from *entry-¿name* into *file_name* so that a null character can be appended to it. The inode and name of the file is then displayed.

Finally, the position of the following entry in the list is given by *entry-¿rec_len*. This field indicates the length in bytes of the current entry record. Therefore, the next entry is located at address *(void*) entry + entry-¿rec_len*.

Notice that the code above only works if the size of the directory is less than a block. If the entries list takes more than one block, the program will crash.

ext2list.c lists the contents of the root directory of the floppy disk. question Q8. To speed up deletion of an entry from a directory, the Linux kernel sets the inode field of the entry to be deleted to 0, and suitably increments the value of the rec_len field of the previous valid entry. Redraw the figure above after deletion of the test.txt entry. question Q9. Modify ext2list.c so that it shows the contents of the whole disk

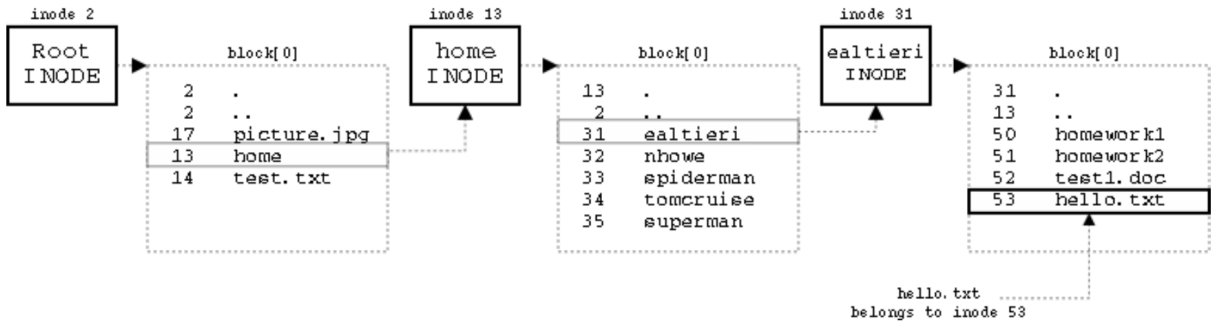
8 Locating a file

Locating the data blocks belonging to a file implies locating its inode in the inode table first. The inode of the desired file is generally not known at the time the open operation is issued. What we know is the path of the file. For example:

```
int fd = open("/home/ealtieri/hello.txt", O_RDONLY);
```

The desired file is hello.txt, while its path is */home/ealtieri/hello.txt*.

To find out the inode belonging to the file we first need to descend through its path, starting from the root directory, until we reach the file's parent directory. At this point we can locate the *ext2_dir_entry_2* entry corresponding to *hello.txt* and then its inode number.



Once the inode of the file is known, the data blocks belonging to the *hello.txt* are specified by the *inode.block[]* array.

The root directory is always located at inode 2.

question Q9. Illustrate the process of looking up the file */usr/local/bin/gcc* similarly to the figure above. You are free to assign any number to the inodes as long as your numbers are consistent and do not exceed the maximum inode number (see *mkfs.ext2*).