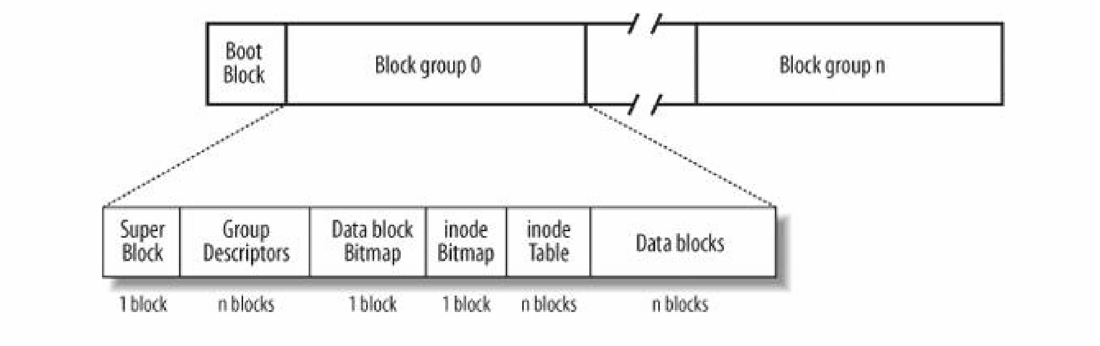
The EXT2 file system

Revised and extended version of (http://sunsite.nus.sg/LDP/LDP/tlk/node95.html)

1. Physical layout of the ext2 file system

The EXT2 file system, like a lot of the file systems, is built on the premise that the data held in files is kept in data blocks. These data blocks are all of the same length and, although that length can vary between different EXT2 file systems the block size of a particular EXT2 file system is set when it is created. Every file's size is rounded up to an integral number of blocks. If the block size is 1024 bytes, then a file of 1025 bytes will occupy two 1024 byte blocks. Unfortunately this means that on average you waste half a block per file. Not all of the blocks in the file system hold data, some must be used to contain the information that describes the structure of the file system. EXT2 defines the file system topology by describing each file in the system with an inode data structure. An inode describes which blocks the data within a file occupies as well as the access rights of the file, the file's modification times and the type of the file. Every file in the EXT2 file system is described by a single inode and each inode has a single unique number identifying it. The inodes for the file system are all kept together in inode tables. EXT2 directories are simply special files (themselves described by inodes) which contain pointers to the inodes of their directory entries.



The figure above shows the layout of the EXT2 file system as occupying a series of blocks in a block structured device. So far as each file system is concerned, block devices are just a series of blocks which can be read and written. A file system does not need to concern itself with where on the physical media a block should be put, that is the job of the device's driver. Whenever a file system needs to read information or data from the block device containing it, it requests that its supporting device driver reads an integral number of blocks[1]. The EXT2 file system divides the logical partition that it occupies into Block Groups. Each group duplicates information critical to the integrity of the file system as well as holding real files and directories as blocks of information and data. This duplication is necessary should a disaster occur and the file system need recovering. The subsections describe in more detail the contents of each Block Group.

[1] In our exercise we just “open” the /dev/fd0 and perform series of size of block byte “read”s.

2. The superblock

The Superblock contains a description of the basic size and shape of this file system. The information within it allows the file system manager to use and maintain the file system. Usually only the Superblock in Block Group 0 is read when the file system is mounted but each Block Group contains a duplicate copy in case of file system corruption. Amongst other information it holds the:

Magic Number
: This allows the mounting software to check that this is indeed the Superblock for an EXT2 file system.

Revision Level
: The major and minor revision levels allow the mounting code to determine whether or not this file system supports features that are only available in particular revisions of the file system. There are also feature compatibility fields which help the mounting code to determine which new features can safely be used on this file system,

Mount Count and Maximum Mount Count
: Together these allow the system to determine if the file system should be fully checked. The mount count is incremented each time the file system is mounted and when it equals the maximum mount count the warning message ``maximal mount count reached, running e2fsck is recommended" is displayed,

Block Group Number
: The Block Group number that holds this copy of the Superblock,

Block Size
: The size of the block for this file system in bytes, for example 1024 bytes,

Blocks per Group
: The number of blocks in a group. Like the block size this is fixed when the file system is created,

Free Blocks
: The number of free blocks in the file system,

Free Inodes
: The number of free Inodes in the file system,

First Inode
: This is the inode number of the first inode in the file system. The first inode in an EXT2 root file system would be the directory entry for the '/' directory.

Above information is contained in the struct ext2_super_block. Partial list of the ext2_super_block fields are depicted bellow:

```
struct ext2_super_block {
        __le32  s_inodes_count;         /* Inodes count */
        __le32  s_blocks_count;         /* Blocks count */
        __le32  s_r_blocks_count;       /* Reserved blocks count */
        __le32  s_free_blocks_count;    /* Free blocks count */
        __le32  s_free_inodes_count;    /* Free inodes count */
        __le32  s_first_data_block;     /* First Data Block */
        __le32  s_log_block_size;       /* Block size */
        __le32  s_log_frag_size;        /* Fragment size */
        __le32  s_blocks_per_group;     /* # Blocks per group */
        __le32  s_frags_per_group;      /* # Fragments per group */
        __le32  s_inodes_per_group;     /* # Inodes per group */
        __le32  s_mtime;                /* Mount time */
        __le32  s_wtime;                /* Write time */
        __le16  s_mnt_count;            /* Mount count */
        __le16  s_max_mnt_count;        /* Maximal mount count */
        __le16  s_magic;                /* Magic signature */
        __le16  s_state;                /* File system state */
        __le16  s_errors;               /* Behaviour when detecting errors */
        __le16  s_minor_rev_level;      /* minor revision level */
        __le32  s_lastcheck;            /* time of last check */
        __le32  s_checkinterval;        /* max. time between checks */
        __le32  s_creator_os;           /* OS */
        __le32  s_rev_level;            /* Revision level */
        __le16  s_def_resuid;           /* Default uid for reserved blocks */
        __le16  s_def_resgid;           /* Default gid for reserved blocks */
```

Other fields are irrelevant to us right now. Use the #include <linux/ext2_fs.h> preprocessor directive in order to obtain the struct ext2_super_block definition. __le16, __le32 are defined in <linux/types.h> (includes path on our Knoppix machines is located in /usr/include) and denote the little-endian ordering for words and double-words (the least significant byte is stored at the highest address).

The `s_inodes_count` field stores the number of inodes, while the `s_blocks_count` field stores the number of blocks in the Ext2 filesystem.

The `s_log_block_size` field expresses the block size as a power of 2, using 1,024 bytes as the unit. Thus, 0 denotes 1,024-byte blocks, 1 denotes 2,048-byte blocks, and so on.

The `s_blocks_per_group` and `s_inodes_per_group` fields store the number of blocks and inodes in each block group, respectively.

3. Mapping disk data directly to structures

For this exercise, you are to use devise interface as thought it is a simplified block device interface. That is, you are to open the file ("/dev/fd0") and then read or write its contents as full ext2 blocks. Use the following functions to read the disk.

```
int fid; /* global variable set by the open() function */

int block_size;     /* bytes per sector from disk geometry */

…

fid = open ("/dev/fd0", O_RDWR);

block_size = /* read from the superblock */



int fd_read(int block_number, char *buffer){

        int dest, len;

        dest = lseek(fid, block_number * block_size, SEEK_SET);

        if (dest != block_number * block_size){

             /* Error handling */

        }

        len = read(fid, buffer, block_size);

        if (len != block_size){

                /* error handling here */

        }

        return len;

}
```

Then copy from buffer to desired structure. See, for example, how to read in to the super block structure:

```
struct ext2_super_block sb;

memcpy(sb, buffer, sizeof(struct ext2_super_block);
```

4. The EXT2 Group Descriptor

Each Block Group has a data structure describing it. Like the Superblock, all the Group Descriptors for all of the Block Groups are duplicated in each Block Group in case of file system corruption.

Each Group Descriptor contains the following information:

Blocks Bitmap
: The block number of the block allocation bitmap for this Block Group. This is used during block allocation and deallocation,

Inode Bitmap
: The block number of the inode allocation bitmap for this Block Group. This is used during inode allocation and deallocation,

Inode Table
: The block number of the starting block for the inode table for this Block Group. Each inode is represented by the EXT2 inode data structure described below.

Free blocks count, Free Inodes count, Used directory count

The group descriptors are placed on after another and together they make the Group Descriptors Table. Each Blocks Group contains its copy of the entire Table of Group Descriptors after its copy of the Superblock. Only the first copy (in Block Group 0) is actually used by the EXT2 file system. The other copies are there, like the copies of the Superblock, in case the main copy is corrupted.

An `ext2_group_desc` (from /usr/include/linux/ext2_fs.h) corresponds for Group Descriptor structure.

```
struct ext2_group_desc
{
        __le32  bg_block_bitmap;                /* Blocks bitmap block */
        __le32  bg_inode_bitmap;                /* Inodes bitmap block */
        __le32  bg_inode_table;         /* Inodes table block */
        __le16  bg_free_blocks_count;   /* Free blocks count */
        __le16  bg_free_inodes_count;   /* Free inodes count */
        __le16  bg_used_dirs_count;     /* Directories count */
```

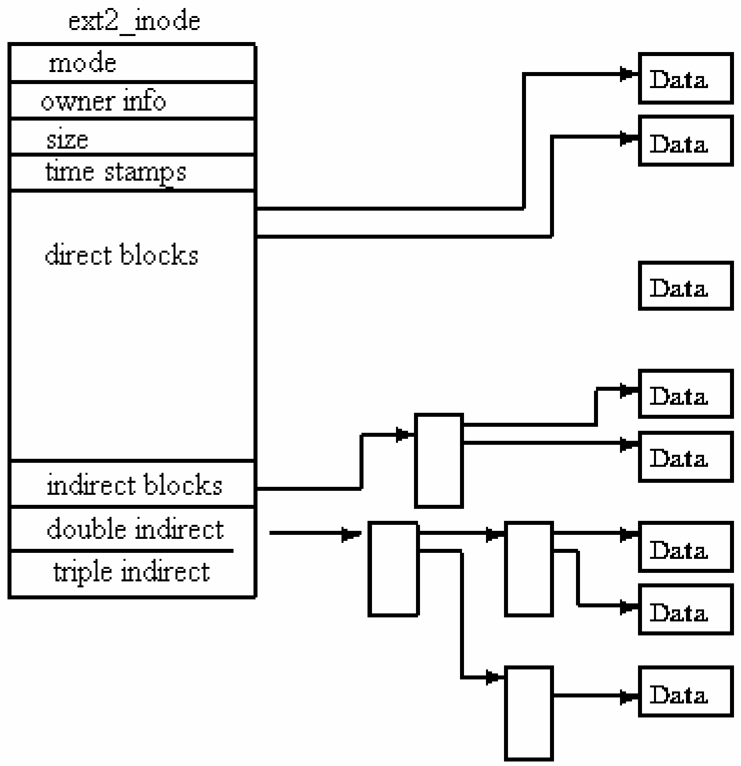
The `bg_free_blocks_count`, `bg_free_inodes_count`, and `bg_used_dirs_count` fields are used when allocating new inodes and data blocks. These fields determine the most suitable block in which to allocate each data structure.

5. Data block bitmap and inode bitmap

The bitmaps are sequences of bits, where the value 0 specifies that the corresponding inode or data block is free and the value 1 specifies that it is used. Because each bitmap must be stored inside a single block and because the block size can be 1,024, 2,048, or 4,096 bytes, a single bitmap describes the state of 8,192, 16,384, or 32,768 blocks.

6. Inode table

In the EXT2 file system, the inode is the basic building block; every file and directory in the file system is described by one and only one inode. The EXT2 inodes for each Block Group are kept in the inode table together with a bitmap that allows the system to keep track of allocated and unallocated inodes. Figure bellow shows the EXT2 inode:



Amongst other information, it contains the following fields:

mode

This holds two pieces of information; what does this inode describe and the permissions that users have to it. For EXT2, an inode can describe one of file, directory, symbolic link, block device, character device or FIFO.

Owner Information

The user and group identifiers of the owners of this file or directory. This allows the file system to correctly allow the right sort of accesses,

Size

The size of the file in bytes,

Timestamps

The time that the inode was created and the last time that it was modified,

Datablocks

Pointers to the blocks that contain the data that this inode is describing. The first twelve are pointers to the physical blocks containing the data described by this inode and the last three pointers contain more and more levels of indirection. For example, the double indirect blocks pointer points at a block of pointers to blocks of pointers to data blocks. This means that files less than or equal to twelve data blocks in length are more quickly accessed than larger files.

An `ext2_inode` (from linux/ext2_fs.h) corresponds for Inode structure:

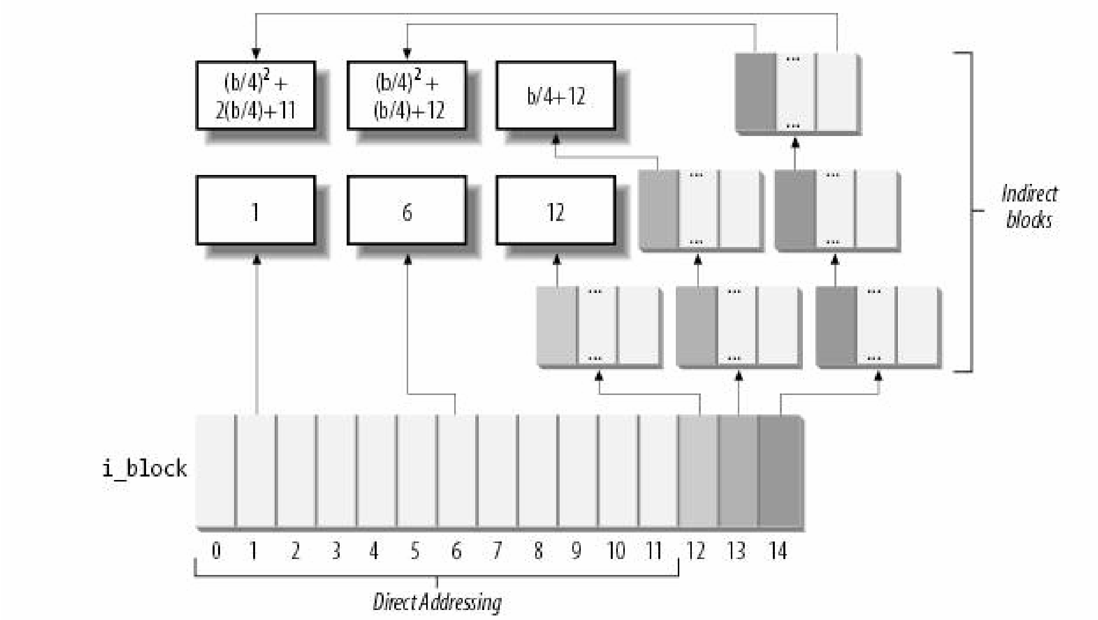
```
struct ext2_inode {
        __le16  i_mode;         /* File mode */
        __le16  i_uid;          /* Low 16 bits of Owner Uid */
        __le32  i_size;         /* Size in bytes */
        __le32  i_atime;        /* Access time */
        __le32  i_ctime;        /* Creation time */
        __le32  i_mtime;        /* Modification time */
        __le32  i_dtime;        /* Deletion Time */
        __le16  i_gid;          /* Low 16 bits of Group Id */
        __le16  i_links_count;  /* Links count */
        __le32  i_blocks;       /* Blocks count */
        __le32  i_flags;        /* File flags */
        union {
                struct {
                        __le32  l_i_reserved1;
                } linux1;
                struct {
                        __le32  h_i_translator;
                } hurd1;
                struct {
                        __le32  m_i_reserved1;
                } masix1;
        } osd1;                         /* OS dependent 1 */
        __le32  i_block[EXT2_N_BLOCKS];/* Pointers to blocks */
```

`osd1` union field that comes after the `i_flags` is irrelevant to us right now (it contains specific operating system information).

The `i_size` field stores the effective length of the file in bytes, while the `i_blocks` field stores the number of data blocks that have been allocated to the file.

The values of `i_size` and `i_blocks` are not necessarily related. Because a file is always stored in an integer number of blocks, a nonempty file receives at least one data block and `i_size` may be smaller than `(size of block)` * `i_blocks`. On the other hand, by applying `lseek` a file may contain holes. In that case, `i_size` may be greater than `(size of block)` * `i_blocks`.

The `i_block` field in the disk inode is an array of `EXT2_N_BLOCKS` components that contain logical block numbers. In the following discussion, we assume that `EXT2_N_BLOCKS` has the default value, namely 15. The array is illustrated in:

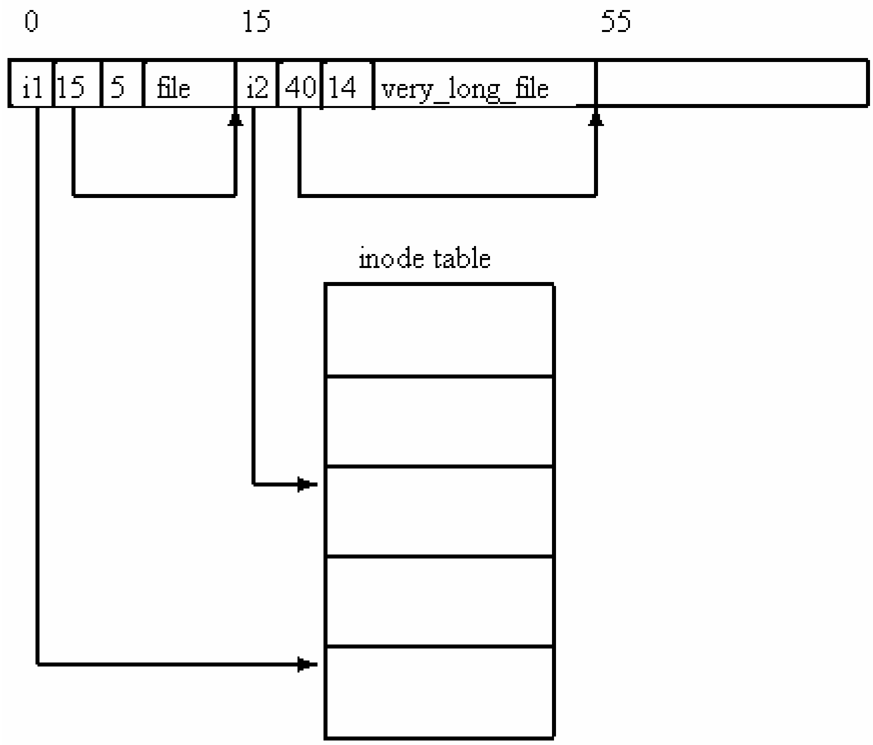


As can be seen in the figure, the 15 components of the array are of 4 different types:

- The first 12 components yield the logical block numbers corresponding to the first 12 blocks of the file to the blocks that have file block numbers from 0 to 11.
- The component at index 12 contains the logical block number of a block, called indirect block, that represents a second-order array of logical block numbers. They correspond to the file block numbers ranging from 12 to b/4+11, where b is the filesystem's block size (each logical block number is stored in 4 bytes, so we divide by 4 in the formula). Therefore, the one must look in this component for a pointer to a block, and then look in that block for another pointer to the ultimate block that contains the file contents.
- The component at index 13 contains the logical block number of an indirect block containing a second-order array of logical block numbers; in turn, the entries of this second-order array point to third-order arrays, which store the logical block numbers that correspond to the file block numbers ranging from b/4+12 to $(b/4)^2+(b/4)+11$.
- Finally, the component at index 14 uses triple indirection: the fourth-order arrays store the logical block numbers corresponding to the file block numbers ranging from $(b/4)^2+(b/4)+12$ to $(b/4)^3+(b/4)^2+(b/4)+11$.

7. EXT2 Directories

In the EXT2 file system Directories are special files that are used to create and hold access paths to the files in the file system. This figure shows the layout of a directory entry in memory:



A directory file is a list of directory entries, each one containing the following information:

inode
: The inode for this directory entry. This is an index into the array of inodes held in the Inode Table of the Block Group. In figure, the directory entry for the file called `file` has a reference to inode number `i1`,

name length
: The length of this directory entry in bytes,

name
: The name of this directory entry.

The first two entries for every directory are always the standard ``." and ``.." entries meaning ``this directory" and ``the parent directory" respectively.

An `ext2_dir_entry_2` (from linux/ext2_fs.h) corresponds for directory structure:

```
struct ext2_dir_entry_2 {
        __le32  inode;                /* Inode number */
        __le16  rec_len;              /* Directory entry length */
        __u8    name_len;             /* Name length */
        __u8    file_type;
        char    name[EXT2_NAME_LEN];  /* File name */
};
```

The structure has a variable length, because the last `name` field is a variable length array of up to `EXT2_NAME_LEN` characters (usually 255). Moreover, for reasons of efficiency, the length of a directory entry is always a multiple of 4 and, therefore, null characters (`\0`) are added for padding at the end of the filename, if necessary. The `name_len` field stores the actual filename length.

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

The `file_type` field stores a value that specifies the file type. The `rec_len` field may be interpreted as a pointer to the next valid directory entry: it is the offset to be added to the starting address of the directory entry to get the starting address of the next valid directory entry. To delete a directory entry, it is sufficient to set its `inode` field to 0 and suitably increment the value of the `rec_len` field of the previous valid entry. **Read the `rec_len` field of carefully; you'll see that the *oldfile* entry was deleted because the `rec_len` field of *usr* is set to 12+16 (the lengths of the *usr* and *oldfile* entries).**

8. Finding a Directory/File in an EXT2 File System

A Linux filename has the same format as all Unix filenames have. It is a series of directory names seperated by forward slashes (``/") and ending in the file's name. Let's call them **path components** or **direntries**. One example filename would be `/home/rusling/.cshrc` where `/home` and `/rusling` are directory names and the file's name is `.cshrc`. Like all other Unix systems, Linux does not care about the format of the filename itself; it can be any length and consist of any of the printable characters. To find the inode representing this file within an `EXT2` file system the system must parse the filename a directory at a time until we get to the file itself.

The first inode that we need is the inode for the root of the file system and we find its number in the file system's superblock. To read an EXT2 inode we must look for it in the inode table of the appropriate Block Group. If, for example, the root inode number is 2 then we need the 2nd inode from the inode table of Block Group 0. The root inode is for an EXT2 directory, in other words the mode of the root inode describes it as a directory and it's data blocks contain EXT2 directory entries.

`home` is just one of the many directory entries and this directory entry gives us the number of the inode describing the `/home` directory. We have to read this directory (by first reading its inode and then reading the directory entries from the data blocks described by its inode) to find the `rusling` entry which gives us the number of the inode describing the `/home/rusling` directory. Finally we read the directory entries pointed at by the inode describing the `/home/rusling` directory to find the inode number of the `.cshrc` file and from this we get the data blocks containing the information in the file.

9. Approaching the maman13

1) Assuming that the boot block is of 1024 bytes, read the superblock from the block group 0 and make sure you read its content correctly (e.g. you should find out that the number of block groups on /dev/fd0 is 1, the size of block is 1024 bytes, the number of inodes is 184).

2) According to the filesystem layout found in the superblock, compute n – the number of blocks assigned for the group descriptors (look again at the figure in paragraph "***Physical layout of the ext2 file system***").

3) Calculate the starting point of the Inode table.

4) Write a subroutine that given an inode number returns the corresponding inode from the inode table.

5) Write a subroutine that given a path component (as a string) and an inode number of directory file, searches the blocks of the directory file to see if the directory file

contains the given path component. If found, corresponding `struct ext2_dir_entry_2` is returned.

6) Write a subroutine that given a full path finds whether the path corresponds to a valid directory on the /dev/fd0 (use the subroutine from the step 5).

7) Write a subroutine that given a full path of a valid directory on the /dev/fd0 prints the content of the directory. One way to accomplish this goal is by changing the subroutine from the step 5 (add additional parameter – a pointer to a pretty_print function that prints the information considering the direntries).

8) Use the subroutine 6 to implement my_cd from maman 13.

9) Use the subroutine 7 to implement my_dir from maman 13.