Middle East Technical University | Department of Computer Engineering

CENG 334

Operating Systems

Spring 2018-2019

Take Home Exam 3

Filesystems

Due date: **20 May 2019 23:55**

1 Objectives

The goal of the assignment is to familiarize you with filesystem structures. Towards this end, you will write a program that can read a regular file and copy its content as a new file into an ext2 image (a file containing the contents and structure of a disk volume or an entire data storage device), without mounting it.

2 ext2 filesystem

The structure of the **ext2 filesystem**, shown in Figure 1, is as follows; The first 1024 bytes of the disk is reserved as a *boot block*. This block is followed by a number of *block/block groups*. Each block group starts with its *super block*, followed by *group descriptors*. The *blocks bitmap* and *inodes bitmap* structures store information about free/allocated blocks and free/allocated inodes in the group. Each bitmap structure takes 1 block and hence the block and inode bitmaps occupy 2 blocks. The *inode table*, size of which depends on the number of inodes created during the formatting process, is stored in subsequent blocks. The remaining blocks within the group are used as *data blocks*.

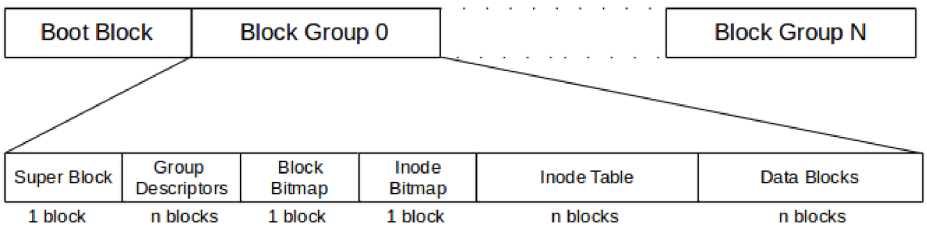

Figure 1: ext2 filesystem structure.

Here we provide some conventions about the ext2 filesystem, to ease your introduction:

- Block numbering starts from 1 (not zero!).
- Block numbering starts at the beginning of the disk.
- The super block of the first group (namely Block/block group 0 in Figure 1) resides in block 1.
- inode numbering starts from 1 (not zero!).
- The root directory inode always resides in inode number 2.
- The first 11 inodes are reserved.
- There is always a lost+find directory in the root directory.
- The total number of data and inode blocks and number of inode and data blocks in each block group are defined in superblock.

The *super block* structure is defined as follows in the Linux kernel:

```
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 */
    __le32  s_first_ino;         /* First non-reserved inode */
    __le16   s_inode_size;       /* size of inode structure */
    /* there are other irrelevant fields */
};
```

where `__le32` is 32 bit unsigned integer and `__le16` is 16 bit unsigned integer.

Note that;

- `s_blocks_count` is the total number of blocks in the image
- `s_inodes_count` is the total number of inodes in the image
- $2^{\texttt{s\_log\_block\_size}+10}$ is the block size. A typical value is 1024.

- `s_inode_size` is the size of an inode structure. A typical value is 128.
- The number of block groups can be calculated as ceiling of $\frac{\texttt{s\_blocks\_count}}{\texttt{s\_blocks\_per\_group}}$
- inode's are numbered globally across the whole volume (or image) as opposed to within the block group.
- The block group in which an inode resides can be computed using the $\left\lfloor \frac{(\texttt{inode}-1)}{\texttt{s\_inodes\_per\_group}} \right\rfloor$ formula.
- Within the inode table of a block group (table address is given in group descriptor table entry), a modulo operation determines the index of the inode.
- The size of an inode is smaller than the size of a block, hence the lookup process should take the inode size into account.
- Data block addresses in inode structure are absolute block addresses of the filesystem image.
- The block group containing a data block can be calculated as $\left\lfloor \frac{\texttt{blockaddr}}{\texttt{s\_blocks\_per\_group}} \right\rfloor$. Remainder of this division gives index of the data block within the block group, which is used in locating its index in block allocation table.
- Each block group may contain a superblock backup copy, group descriptor table backup copy. Groups 0, 1 and groups that are powers of 3, 5, or 7 contains these copies. The others directly start with the bitmap data. As a result, the offset of inode and data block bitmaps are not fixed. You need to read **group descriptor table** for offset of bitmap blocks and inode table.

The *group descriptor table* blocks, which comes after each super block copy, describes the structure of each block group:

```
struct group_descriptor {
    __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 */
    __le16  bg_pad;
    __le32  bg_reserved[3];
} group_descriptors[NGROUPS];
```

In the table, there is an entry for each group, which contains the absolute block addresses of *data blocks bitmap*, *inodes bitmap*, and *inodes table*. The number of blocks that the blocks/inodes bitmap occupy can be computed by dividing the number of blocks/inodes by 8 (i.e number of bits in a byte) rounded up to the block size. For instance, one 1024 byte block is sufficient for the *blocks bitmap* to store the free/allocated information about 8192 blocks.

An inode table has the following 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 */
    __le32  i_reserved;  /* OS dependent reserved */
    __le32  i_block[12];/* Pointers to blocks */
    __le32  i_ind_block;
    __le32  i_dind_block;
    __le32  i_tind_block;
    /* rest is irrelevant, pad it to 128 bytes */
} inode_table[N_INODES_PER_BLOCK];
```

A directory is represented as a file and hence has a corresponding inode which contains direct/indirect pointers to its data blocks. The data blocks of a directory stores *directory entry* structures as follows:

```
struct ext2_dir_entry {
    __le32  inode;          /* Inode number */
    __le16  rec_len;        /* Directory entry length */
    __le16  name_len;       /* Name length */
    char    name[];         /* File name, up to EXT2_NAME_LEN */
};
```

where `rec_len` is the length of the whole entry and `name_len` is the length of the file `name` string. `rec_len` is usually rounded up to 4 bytes word.

The data blocks of a directory contains all files (note that directories are also files) under this directory as sequence of these directory entries. A new file entry can be created after the last directory entry, which can be found after a sequential search.

More information about the ext2 file system can be found in the following links:

- `http://www.nongnu.org/ext2-doc/ext2.html`: ext2 documentation.
- `https://wiki.osdev.org/Ext2`: The OSDev wiki page on ext2
- Also `http://web.mit.edu/tytso/www/linux/ext2intro.html`: A page by Dave Poirer containing more details.

2.1 Details

- You can create a disk image with 128 blocks of size 1024 with the following command:

  ```
  $ dd if=/dev/zero of=image.img bs=1024 count=128
  ```

- The disk image can then be formatted with mke2fs. The following commands format the disk and force the creation of 32 inodes.

  ```
  $ mke2fs -N 32 image.img
  ```

- You can mount the image (for verification purposes only) by creating a loopback device by:

  ```
  $ mkdir mnt
  $ sudo mount -o loop image.img mnt
  ```

  and unmount with:

  ```
  $ umount mnt
  ```

- On computers where you do not have administrative privileges (such as the ones in the lab), you can use FUSE based fuseext2 to mount your image to a folder you own as:

```
$ fuseext2 -o rw + image.img mnt
```

and unmount with:

```
$ fusermount -u mnt
```

- `dumpe2fs` tool can be used to inspect structure of filesystem.
- You are expected to copy **only regular files** into the ext2 image. Your implementation should support different block sizes, different number of inodes and different number of blocks in the image. Copying of files other than regular files is not required.
- You can inspect differences between two image files using a **xxd** hex dump and **diff**. For instance, you can use the following command to inspect the differences between two images:

```
$ diff <(xxd image1.img) <(xxd image2.img) > images.diff
```

2.2 Implementation, Compilation & Execution Details

Any standard library can be used in your code. However, **the use of libraries with ext2 implementation is forbidden!**

You have to provide a makefile with your implementation which creates an executable named **filecopy**.

Your code will be tested using the command:

```
$ ./filecopy imagefile sourcefile targetdirpath|targetdirinode
```

which should result in copying the regular file `sourcefile` into the ext2 image file `imagefile`. The file should be created with same name under the `targetdirpath` directory of the ext2 image. `targetdirpath` will be an absolute path. If it does not start with '/', it should be assumed relative to '/', root of the ext2 image. Instead of a directory path, the target can be given as location of an inode block, which is the inode for a directory.

The steps of your task are:

- Open the ext2 image file.
- If `targetdirpath` is given, traverse the path and get the directory data blocks.
- If `targetdirinode` is given get the inode and get the directory data blocks.
- Allocate a new inode, fill in metadata from metadata of `sourcefile` (see `man 2 stat`).
- Allocate data blocks for the new file. For simplicity, assume that the file data will fit only in direct blocks of the inode. if it is larger, it is truncated to direct data block addressing.
- Set data blocks of the inode of the created file.
- Insert an entry in data blocks of the directory for the file mapping file name into file inode.
- Close the image file.

After these steps, the copied file should be visible in the given path of the ext2 image.

You will be given a valid and non corrupted ext2 image file. However note that it does not have to be empty and may already contain files in it. Your code will be tested multiple times with different files and images successively. When copying a file, you are expected to print the inode number of the file that you created and its corresponding data blocks, such as:

```
$ ./filecopy imagefile file targetdirectory
12 8,9,10,11
```

3 Restrictions, Grading and Warnings

- Your implementation should be in C/C++.
- The submissions should be done via COW.
- Create a tar.gz file named **hw3.tar.gz** that contains all your source code files and a makefile.
- The executable should be named **filecopy**.
- The following sequence of commands should compile and execute your code.

```
$ tar -xf hw3.tar.gz
$ make all
$ ./filecopy imagefile sourcefile targetdirectory
$ ./filecopy imagefile sourcefile targetinode
```

- Your codes will be evaluated through a black-box approach and have to compile and run on lab. machines.
- After your code runs, the target image filesystem check should not produce any integrity errors.
- The minimal implementation for getting partial points from this homework is the correct implementation of:
  `filecopy imagefile sourcefile targetinode`.
  which is worth 60 points.
- If `sourcefile` is directory, and recursive copy of the directory is implemented, a **15 point bonus** will be granted.
- If files larger than 12 direct blocks is supported (up to triple indirect) a **15 point bonus** will be granted.
- Please ask your questions related to the homework on **piazza only** so that the discussions become public and would benefit you all.
- **Cheating: We have zero tolerance policy for cheating**. Sharing part or whole of your code with other students, or copying code from other sources on the internet will be considered as cheating, and disciplinary action will be taken against.