Project 5: File systems

Due May 8 by 9pm **Points** 120

Objectives

To get experience with file system structure

To get acquainted with file system utilities such as mkfs and debugfs

*To help the police catch the bad guy

Updates

4/23: the necessary starter source code can now be found in `/p/course/cs537-swift/tests/p5`

Overview

This is a group project that can be done in pairs of 2. In this project, you will recover all image files from an ext2 disk image. One of you should work on the content recovery of the jpg images (the first part), and the other one should work on image file name recovery (the second part).

1) Project Description

Recently there have been lots of bank robberies. Just a few days ago, the police have identified a possible subject. The subject's motor vehicle contained, among other items, some hard drives (without the laptop). Apparently, the subject had deleted the files without reformatting the drives. Fortunately, the subject had never taken CS 537 before. This means that the subject does not know that most data and indeed most of the file control blocks still reside on the drives.

The police know that bank robbers usually take pictures of banks that they plan to rob. Thus, the police hire you, a file system expert, to be part of the forensics team attempting to reconstruct the contents of the disks. Each disk will be given to you in the form of a **disk image (http://en.wikipedia.org/wiki/Disk_image)** . A disk image is simply a file containing the complete contents and file system structures (For more, see "Ext2 image file" section below).

To catch the bad guy and prevent future robberies, your goal is to reconstruct all pictures (jpg files only) in the disk images and make the subject regret not taking CS 537 in the past. Of course, you may understand the regret of taking 537.

2) Project Specification

You need to write a program called `runScan` that takes **two arguments**, an input file that contains the disk image and an output directory where your output files will be stored. The tasks of your program

are as follows:

First, you need to reconstruct **all** jpg files (both undeleted and deleted ones) from a given disk image. To do this, you need to scan all inodes that represent regular files and check if the first data block of the inode contains the jpg magic numbers: `FF D8 FF E0` or `FF D8 FF E1 or FF D8 FF E8`. (For more about file signatures, you can visit **this page** (http://www.garykessler.net/library/file_sigs.html)). For example, if you read the first data block of a file and put it in `char buffer [1024]` , then this code will identify whether it is a jpg file or not:

```
int is_jpg = 0;
if (buffer[0] == (char)0xff &&
    buffer[1] == (char)0xd8 &&
    buffer[2] == (char)0xff &&
    (buffer[3] == (char)0xe0 ||
     buffer[3] == (char)0xe1 ||
     buffer[3] == (char)0xe8)) {
         is_jpg = 1;
}
```

Once you identify an inode that represents a jpg file, you should copy the content of that file to an output file (stored in your 'output/' directory), using the inode number as the file name. For example, if you detect that inode number 18 is a jpg file, you should create a file 'output/file-18.jpg' which will contain the exact data reachable from inode 18. (See the Example section below for more).

The second part of your task is to find out the **filenames** of those inodes that represent the jpg files. Note that filenames are not stored in inodes, but in directory data blocks. Thus, after you get the inode numbers of the jpg files, you should scan all directory data blocks to find the corresponding filenames. After you get the filename of a jpg file, you should again copy the content of that file to an output file, but this time, using the actual filename. For example, if inode number 18 is 'bank-uwcu.jpg', you should create a file 'output/bank-uwcu.jpg' which will be identical to 'output/file-18.jpg'.

In summary, in your final output directory, for each jpg file, there should be a file with an inode number as the filename, and another file with the same filename as the actual one.

When your program starts, your program should create the specified output directory. If the directory already exists, print an error message (any message) and exit the program. You can know if a directory exists by using the `opendir(name)` (https://linux.die.net/man/3/opendir) system call.

3) Source code and disk images

We provide a header file that specifies some necessary ext2 data structures. We also provide source code that parses an ext2 disk image and gives you the locations of the inode tables, and some important information such as the number of inodes in a cylinder group, number of inode tables, etc. Please see updates once they are posted. It will be available at:

```
% /p/course/cs537-swift/tests/p5
```

To get you started, the first disk image (image-01) for you to work on; more disk images from the subject that might contain the interesting images will come later. It has two small jpg files in the root directory, one not deleted (./indonesian-flag-extra-small.jpg) and one that has been deleted (./french-

flag-small.jpg). It also contains a not deleted larger image in a subdirectory (./pics/architecture_big_ben.jpg). The disk image is available here:

```
/p/course/cs537-swift/tests/p5/disk_images
```

4) Tips and Requirements

Example

Here is how your program will be called:

```
% ./runScan anImageDisk outputDirectory
```

For example:

```
% ./runScan image-01 output01
```

After scanning image-01, your program should be able to find all the three jpg files and copy those three files from the disk image to your output directory. As specified above, you should create two copies of each file, one with the inode-number as the filename, and one with the actual filename. Hence, your output directory should look like this:

```
% ls output01/
architecture_big_ben.jpg  14
french-flag-small.jpg     13
indonesian-flag-extra-small.jpg   12
file-13.jpg
file-14.jpg
file-15.jpg
```

You will get a full credit if you copy the correct amount of bytes for each of your output files. In other words, you should reconstruct the jpg files as they were stored in the beginning. To confirm this, you can 'cmp' your output files with the original files (the three jpg files can be obtained from /p/course/cs537-swift/tests/p5`/jpg_images/`). For example:

```
% cmp output01/french-flag-small.jpg /p/course/cs537-swift/tests/p5/jpg_images/french-flag-sma
ll.jpg
```

You should also check that the other three files that are named using the inode numbers are correct. For example, from your program, you can tell that file-18.jpg is extra-large-hsbc.jpg, hence cmp-ing these two files should give a success (i.e. return nothing).

```
% cmp output01/architecture_big_ben.jpg output01/file-14.jpg
```

Ext2 disk image file

What is an ext2 disk image file? An ext2 disk image file is actually a file system that you can mount. Unfortunately, you can mount a file system only if you have root access. If you do, you can mount an

ext2 disk image (e.g. image-01) to a subdirectory (e.g. mnt/):

```
% mount image-01 mnt -o loop
(Note: You cannot run this in the CS lab, because you are not the root)
```

However, there is another way you can play around with the disk image: use **debugfs (http://lwn.net/Articles/115405/)** .

Debugfs can be found in /sbin/debugfs. If /sbin is not in your default path, please add this line to your .cshrc file:

```
set path = ( $path /sbin )
```

Debugfs is a basically a tool that can traverse the ext2 file system stored in a disk image. Inside debugfs prompt, you can type commands such as 'ls', 'cd', and many others. To see the full list of supported commands, please check the debugfs manual (man debugfs). Here's an example of a debugfs run:

```
% debugfs image-01
debugfs 1.45.5 (07-Jan-2020)
debugfs:  ls
      2   40755 (2)      0      0    1024 23-Apr-2022 01:24 .
      2   40755 (2)      0      0    1024 23-Apr-2022 01:24 ..
     11   40700 (2)      0      0   12288 23-Apr-2022 01:24 lost+found
     12   40755 (2)      0      0    1024 23-Apr-2022 01:25 pics
     13  100644 (1)      0      0     775 23-Apr-2022 01:26 indonesian-flag-extra-small.jpg
     15   40755 (2)      0      0    1024 23-Apr-2022 01:58 temp
debugfs:  quit
```

As you can see, the root directory of the disk image has three directories (lost+found, temp, and pics). With debugfs, you can traverse these directories (e.g. cd pics). Also note that debugfs only shows only the image file that the robber has not deleted (indonesian-flag-extra-small.jpg). Since debugfs can see thes file, your scanner should also be able to see the file. However, there is one jpg file that the robber has deleted (french-flag-small.jpg). You can use `ls -d` to show the deleted files, and your scanner should find this second file whose bytes still exist in the disk image. You can also use cat or dump to inspect entry contents, such as file names under a directory. Type q to quit file listing.

Test files

To stress-test your program, you should create your own test files. (You do not need to submit your test files; they are for your own benefit). You can create a **real** ext2 disk image just like the first disk image that we give you. First, you need to create a file that will hold the disk image:

```
% dd if=/dev/zero of=your-image bs=30M count=1 seek=0
```

The above command will create a 30MB-file named 'your-image'. Next you need to format this disk image as an ext2 image:

```
% mkfs.ext2 -b 1024 -F your-image
```

The '-b' option specifies that the block size of the disk image is 1024 bytes. The mkfs command will dump some information about the created file system (e.g. how many block groups, inodes, and blocks exist in the system). Now, you have a disk image ready to use. To copy some files to your disk image, run debugfs with write-mode (-w). Inside the debugfs prompt, you can make directories (mkdir) and you can copy files from the local file system to your disk image (write src dst). After you create these files in your disk image, you should type 'quit' so that these files will be fully stored in the disk image. Here's an example on how to copy a jpg file from a local '/tmp' directory to '/mypics' directory inside the disk image:

```
% debugfs -w your-image

debugfs: mkdir mypics

debugfs: cd mypics

debugfs: write /tmp/a.jpg a.jpg
Allocated inode: 13

debugfs: ls
12 (12) .
2 (12) ..
13 (1000) a.jpg

debugfs: quit
```

To delete a file (so that you can test if your scanner can recover the file), use the remove command (rm) inside debugfs.

```
% debugfs -w your-image
debugfs: cd mypics
debugfs: rm a.jpg
debugfs: quit
```

Remember that whenever you have modified your disk image, even just a little bit, always quit debugfs first. This ensures that your modification is reflected in the disk image (rather than being reflected in the memory only). For example, if you want to create and delete a file, you should run debugfs, create the file, quit debugfs, then run debugfs again, delete, and quit again.

Understanding Ext2 on-disk data structures

There are many sources out there that explain Ext2 on-disk data structures in great details. For this project, you only need to know some of the structures. *It is your job to figure out how you use this knowledge to achieve the goals above.*

- **Inode Table:** An ext2 file system is partitioned into multiple cylinder/block groups. In each block group, there is an inode table. The inode table is basically an array of inodes. The number of inodes in an inode table (i.e. the number of inodes in a block group) is specified in the superblock. For this project, we already wrote the parser that gives the locations of the inode tables and the number of inodes in a group. You simply use this information to scan all the inodes.

The inode table does not store inode numbers, hence you should track the inode numbers. Inode number starts from 1. If there are 100 inodes per group, then the first inode table holds inode number 1 to 100, the second holds 101 to 200, and so on. In other words, the first inodeTable[0..99] array stores inodes numbered from 1 to 100.

- **Inode:** An on-disk inode is stored as a 128-bytes **ext2_inode (https://pages.cs.wisc.edu/~remzi/Classes/537/Spring2009/Projects/inode.html)** structure. As you can tell from the structure, an inode stores lots of information such as block pointers, file mode, time, access control list, and user identifications. Of all this information, you only need to know four of them:

inode->i_mode: This field stores information about the mode of the file. An inode can represent a regular file, a directory, a symbolic link, and other file types. For this project, you only care whether an inode is a regular file or a directory. There are two calls that tell you if an inode represents a regular file or a directory:

```
if (S_ISDIR(inode->i_mode)) {
// this inode represents a directory
}
else if (S_ISREG(inode->i_mode)) {
// this inode represents a regular file
}
else {
// this inode represents other file types
}
```

See a use case of two functions can be found in the source code that we give you.

inode->i_size: This field stores the size of the regular file that this inode represents. You need this field to know how many data blocks to read, and how many exact bytes you need to copy from the data blocks to your output file.

inode->i_links_count: This field identifies whether this inode is used or not. If the value is 0, then this inode is a free inode. If the value is not 0, then this inode is being used. For example, if the value is 0 and this inode is a regular file, this implies that the file has been deleted by the user, and will not appear in the file system tree. However, your scanner still can traverse and get the data blocks.

inode->i_block[]: This array stores 12 direct pointers (block numbers of file/directory data blocks), an indirect pointer, a double indirect pointer, and a triple indirect pointer. A pointer is basically a block number. For example, if a file is 14.5 KB long, this file needs to use 15 data blocks. Let's say the data blocks for this file are stored from block 1001 to 1015. The i_block field of this inode looks like this:

```
i_block[0] = 1001;
i_block[1] = 1002;
... i_block[11] = 1012;
i_block[12] = 2000; // the last 3 blocks are stored in indirect block
i_block[13] = 0; // double indirect pointer is not used
i_block[14] = 0; // triple indirect pointer is not used
```

To find the locations of the last 3 blocks, you need to read the indirect block pointed by the indirect pointer (i.e. block 2000). Since a block for this project is always 1024 bytes long, and a pointer (a

block number) is 4 bytes long, then an indirect block simply contains an array of 256 pointers. Thus, the content of block 2000 looks like this:

> offset 0: 1013
> offset 4: 1014
> offset 8: 1015
> offset 12: 0 // unused
> offset 16: 0 // unused
> ...

- **Directory entry:** If an inode represents a directory, then its data blocks contain directory entries. In ext2, directory entries are managed as list of variable length entries. Each directory entry contains the inode number (4 bytes), the length of the entry (2 bytes), the length of the filename (2 bytes), and the filename. Although, the **ext2_dir_entry (https://pages.cs.wisc.edu/~remzi/Classes/537/Spring2009/Projects/direntry.html)** structure specifies that the name is 255 bytes long, it does not use all the 255 bytes to save space.

```
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 */
};
```

For example, let's say in the root directory (inode number 2) of a file system, originally there were four files ('newfile', 'usr', 'oldfile', and 'note'), but 'oldfile' has just been deleted. And, let's say the first data block of this root inode is located in block 3000 (i.e. inode->i_block[0] = 3000). Then, the content of block 3000 looks like this:

```
// each 'x' represents a byte

           inode   | ent | nm. |
           number  | len | len | name
           x x x x | x x | x x | x x x x x x x x
-----------------------------------------------------
offset 00:   2     |  12 |   1 | . \0\0\0
offset 12:   2     |  12 |   2 | . . \0\0
offset 24:   24    |  16 |   7 | n e w f i l e \0
offset 40:   39    |  28 |   3 | u s r \0
offset 52:   41    |  16 |   7 | o l d f i l e \0
offset 68:   43    | 956 |   4 | n o t e
```

Each directory entry is always 4-byte aligned. Thus null-characters are padded in entry that is not 4-byte aligned. The first two entries in a directory data block are always '.' and '..'. The final important note is, although a file has been deleted (e.g. 'oldfile'), the corresponding directory entry might still be around in the disk image. However, this entry will not appear to the user because the record length of the previous entry ('usr') has covered the deleted entry. However, your scanner should be able to see the deleted entries. Hence, your goal for this project is to scan these entries, find the filenames of the both undeleted and deleted pictures.

Since entry length and name length are 2 bytes long, and name does not always end with a null-character, below is one way to parse a directory entry. In this example, we want to parse the directory entry at offset 68.

```
dentry = (struct ext2_dir_entry*) & ( buffer[68] );

int name_len = dentry->name_len & 0xFF; // convert 2 bytes to 4 bytes properly

char name [EXT2_NAME_LEN];
strncpy(name, dentry->name, name_len);
name[name_len] = '\0';

printf("Entry name is --%s--", name);
```

- Other sources
  - Original Paper: **http://e2fsprogs.sourceforge.net/ext2intro.html (http://e2fsprogs.sourceforge.net/ext2intro.html)**
  - **(http://e2fsprogs.sourceforge.net/ext2intro.html)** Read "ext2 data structures": **https://en.wikipedia.org/wiki/Ext2** **(https://en.wikipedia.org/wiki/Ext2)**
  - More on ext2 overview: **(https://github.com/torvalds/linux/blob/master/fs/ext2/ext2.h)** **https://www.kernel.org/doc/html/latest/filesystems/ext2.html (https://www.kernel.org/doc/html/latest/filesystems/ext2.html#options)**
  - Linux ext2 header file: **https://github.com/torvalds/linux/blob/master/fs/ext2/ext2.h (https://github.com/torvalds/linux/blob/master/fs/ext2/ext2.h)**
  - debugfs: **https://www.man7.org/linux/man-pages/man8/debugfs.8.html (https://www.man7.org/linux/man-pages/man8/debugfs.8.html)**

Simplifications

For this project, there are many simplifications:

- You can assume a 1-KB (1024 bytes) block size.
- We will not create a directory that uses more than one data block. Hence to find directory entries, you simply need to read and parse the first data block of a directory inode (i.e., read and parse the block pointed by `inode->i_block[0]`).
- We will not create a huge file that uses the third indirect block. Thus, you don't have to read and parse the third indirect block and its sub-trees.

Turnin

Hand in your source code (*.c and *.h), a Makefile, and a README file. Your Makefile should build the executable 'runScan'. If your program does not work perfectly, your README file should explain what does not work and the reason (if you know). The handin directory will be `/p/course/cs537-swift/turnin/p5`

If you are partnering with another student, include a `partner.txt` file in the format as follows. Only one of you need to hand in the code.

```
partner1_name: partner1_cslogin
partner2_name: partner2_cslogin
```

If you are using slip days on this project, you must copy your code into the corresponding slip directory. Each of you will only need to contribute 1/2 of any slip days you use.