CS111 - Project 3A: File System Dump

INTRODUCTION:

Project 3 is expected to involve much more programming than any of the other projects. You will design and implement programs to analyze file systems and diagnose corruption. In part A, you will write a program to read the image of a file system, analyze it, and summarize its contents in several csv files. These csv files will be used in part B, where we will analyze the consistency of file system data structures and diagnose problems.

Part A can be broken into two major steps:

1. Mount the provided image file on your own Linux and explore it with *debugfs(8)*.
2. Write a program to analyze the image file and output a summary to six csv files (describing the super block, cylinder groups, free-lists, inodes, indirect blocks, and directories).

RELATION TO READING AND LECTURES:

This project more deeply explores the file and directory concepts described in Arpaci chapter 39. This project is based on the same EXT2 file system that is discussed in sections 40.2-40.5. Part B of this project goes much deeper than the introductory discussion of integrity presented in sections 42.1-2.

PROJECT OBJECTIVES:

- reinforce the basic file system concepts of directory objects, file objects, and free space.
- reinforce the implementation descriptions provided in the text and lecture.
- gain experience with the examining, interpreting, and processing information in binary data structures.
- gain practical experience with complex data structures in general, and on-disk data formats in particular.
- (in 3B) reinforce the notions of consistency/integrity provided in the text and lecture.

DELIVERABLES:

A single tarball (.tar.gz) containing:

- a single C source module that compiles cleanly (with no errors or warnings).
- a Makefile to build the program and the tarball.
- a README file describing each of the included files, your UID, and any other information about your submission that you would like to bring to our attention (e.g. limitations, features, testing methodology, etc.).

PROJECT DESCRIPTION:

Historically, file systems were almost always been implemented as part of the operating system, running in kernel mode. Kernel code is expensive to develop, difficult to test, and prone to catastrophic failures. Within the past 15 years or so, new developments have made it possible to implement file systems in user mode, improving maintainability … and in some cases delivering even better performance than could be achieved with kernel code. All of this project will be done as user-mode software.

To ensure data security and integrity, disks are generally protected from access by ordinary applications. Linux supports the creation, mounting, checking, and debugging of file systems stored in ordinary files. In this project, we will provide EXT2 file system images in ordinary files. Because they

are in ordinary files (rather than protected disks) you can access/operate on those file system images with ordinary user mode code.
To complete this assignment, you will need to learn a few things:

- *debugfs(8)* (http://man7.org/linux/man-pages/man8/debugfs.8.html)
- *pread(2)*
- csv format
- EXT2 file system (http://www.nongnu.org/ext2-doc/ext2.html)

Step 1 - study a provided file system image

Your program will be tested on multiple (broken) file systems. Here we provide one as an example. SEASnet servers do not support commands like *sudo(8), mount(8), or debugfs(8)*, so to play with our provided file system image, you will have to install a Linux distribution (if you do not have one) on your own computer. For simplicity and convenience, you may choose to install Linux inside a virtual machine, for example VirtualBox (it's free!).
Then download this image file, and mount it onto your **own** Linux, with the following commands:

```
mkdir fs
sudo mount -o loop disk-image fs
sudo chown -R $USER fs
```

… where $USER is your user-name

Note: this file is 200 Mbytes in size.
Now, you can navigate the file system, just like an ordinary directory, with commands like *ls(1)*, *cat(1)*, and *cd(1)*. After you are done with it, you can unmount with the following command:

```
sudo umount fs
```

Before you start writing your C program to interpret the "disk-image" file, you can explore it further using *debugfs(8)* (in your own Linux). You may find many useful commands in its man page. Some particularly helpful ones are: *stats*, *stat*, *bd*, *testi*, and *testb*. If you have problems interpreting parts of the file system, you can use the *debugfs* program to help you understand the correct contents.
To ensure you are correctly interpreting the file system image, we have included many unusual things, which would not likely be caught by an incorrect implementation:

- sparse files
- large files
- allocated data blocks full of zeroes
- unallocated blocks containing valid data
- files with data beyond their length
- files with long names
- files with syntactically strange names
- directories that span multiple blocks and have obsolete entries for deleted files.

Step 2 - write a program to summarize the file system's contents

In this step, you will write a program called **lab3a** that:

- Reads (only) one file system image according to the provided file name. For example, we may run your program with the above file system image using the following command: ./lab3a disk-image
- Analyzes the provided file system image and outputs six csv files to the current directory. The contents of these csv files are described below. Your program **must** output these files with **exactly the same formats** as shown below. We will use sort and diff to compare your csv files with ours, so a different format will make your program fail the test and the content in the output csv file will be treated as error.

Please note that, although you cannot mount the provided image file on departmental servers, your lab3a program should be able to run on departmental servers, just like all previous assignments.

There are to be six csv-format files, each summarizing a different part of the file system. Remember, you can always check your program's output against *debugfs*'s output. All the information required for the summary can be manually found and checked by using debugfs.
The file system images your program examines will, in many cases be damaged (as might result from missing or mis-directed writes). This means that you will have to do some *Defensive Programming* in your analysis program: validating parameters and pointers before you use them. In addition to correctly analyzing the file system and group parameters, inodes, block pointers, and directory entries, your program will also be expected to report (to stderr) any invalid parameters or addresses.

1. super block

A basic set of file system parameters.

fields(9)	format
magic number	hex
total number of inodes	dec
total number of blocks	dec
block size	dec
fragment size	dec
blocks per group	dec
inodes per group	dec
fragments per group	dec
first data block	dec

Sample correct csv output

```
Ef53,51200,204800,1024,1024,8192,2048,8192,1
```

Sanity checking

- •Magic number must be correct
- •Block size must be reasonable (e.g. power of two between 512-64K)
- •Total blocks and first data block must be consistent with the file size
- •Blocks per group must evenly divide into total blocks
- •Inodes per group must evenly divide into total inodes

Sample error messages (to stderr):

```
Superblock - invalid magic: 0xdead
Superblock - invalid block size: 666
Superblock - invalid block count 200000 > image size 50000
Superblock - invalid first block 100000 > image size 50000
Superblock - 20000 blocks, 1050 blocks/group
Superblock - 1000 Inodes, 66 Inodes/group
```

It is important to validate superblock parameters, because they are used to determine the addresses of inodes and to validate block pointers. If a file system fails any of these tests, log an error and exit without any further processing.

1. group descriptor

Information about each cylinder group

fields(7)	format
number of contained blocks	dec
number of free blocks	dec
number of free inodes	dec
number of directories	dec
(free) inode bitmap block	hex
(free) block bitmap block	hex
inode table (start) block	hex

Sample output (just for the first cylinder group)

8192,7913,2024,2,4,3,5

Sanity checking

- Number of contained blocks must be consistent with superblock
- Bitmap and inode starting blocks must be within the group

Sample error messages (to stderr):

```
Group 7: 100000 blocks, superblock says 50000
Group 7: blocks 100000-150000, free Inode map starts at 165000
Group 7: blocks 100000-150000, free block map starts at 165000
Group 7: blocks 100000-150000, Inode table starts at 165000
```

Group descriptor validation is also critical because this table tells us where the inodes and free bit-maps begin. If a group descriptor fails any of these tests, log an error, and do not attempt to dump out bit-maps or inodes that are not within the group.

1. free bitmap entry:

A list of free inodes and free blocks.

One line of output for each free inode or block in the bitmap. No output should be produced for allocated inodes or blocks.

fields(2)	format
block number of the map	hex
Free block/inode number	dec

Sample output (just first five entries)

```
3,275
3,276
3,277
3,278
3,279
```

Sanity checking

- none … since there are no block numbers within the bit-map, there is no trivial way to tell if the map has been corrupted.

1. Inode

Key information for each allocated inode.

An allocated inode has a file type.

fields(11+15)	format
inode number	dec
file type	char[1]
mode	oct
owner	dec
group	dec
link count	dec
creation time	hex
modification time	hex
access time	hex
file size	dec
number of blocks	dec
block pointers * 15	hex

Sample output (first four inodes only)

```
1,?,0,0,0,0,58201197,58201197,58201197,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
2,d,40755,1000,0,5,58201213,58201213,582011bd,3072,3,105,a602,a107,0,0,0,0,0,0,0,0,0,0,
0
```

3,?,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
4,?,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

Sanity checking

- All block numbers should be between the first data block number and the file system size (as indicated in the super block)

It is important to validate all block pointers before we use them:

- If the file type is directory, we will fetch each data block and interpret it as directory entries.
- If the file spills over into indirect blocks, we will fetch the indirect blocks and list the pointers.

Sample error message (to stderr):

```
Inode 407 - invalid block pointer[9]: 3ab403
```

If any block address falls outside the reasonable range, include it in the output, log an error message. But if an out-of-range block is an indirect block or contains directory entries, do not attempt to further interpret its contents.

1. directory entry

The valid/allocated entries in each directory.

An allocated directory entry can be recognized by a non-zero inode number. Any directory entry that has a zero inode number should be assumed to be empty and not included in the output.

The term *parent inode number*, below, refers to the inode number of the directory that contains all of the entries that are being listed.

The term *entry number*, below, is a number that starts at 0 for the first directory entry, and is incremented for each entry that is encountered (whether allocated or not). If the first ten entries are allocated, the next is free, and the last is allocated, the output should describe entries: 0-9, and 11, with no output for entry 10.

fields(6)	format
parent inode number	dec
entry number	dec
entry length	dec
name length	dec
inode number of the file entry	dec
name	string[2]

Sample output (first three directories)

```
2,0,12,1,2,”.”
2,1,1012,2,2,”..”
2,2,20,10,11,”lost+found”
```

Sanity checking

- entry length should be reasonable (e.g. 8-1024 bytes) and fit within the file length
- name length should fit within the entry length
- file entry inode number should be within the super-block specified range (number of inodes)

If a group descriptor fails any of these tests, log an error. If either the name or entry length is unreasonable, stop interpreting this directory:

Sample error message (to stderr):

```
Inode 204, block 3c05 - bad dirent: Inode = 107538
Inode 204, block 3c05 - bad dirent: len = 3040, namelen = 5240
```

1. indirect block entry

These are all the **non-zero** block pointers in an indirect block. The blocks that contain indirect block pointers are included.

fields(3)	format
block number of the containing block	hex
entry number within that block	dec
block pointer value	hex

Sample output (first three entries)

```
1e10f,0,1e110
1e10f,1,1e111
1e10f,2,1e112
```

Sanity checking

- all block numbers should be legal (within super-block specified range)

It is important to validate all block numbers in indirect blocks for the same reason we had to validate block numbers in inodes: because we will follow those pointers to further interpret the file system.

Sample error message to stderr:

```
Indirect block 3c4 - invalid entry[103] = 304c8df
```

If any block number is out of range log an error message. If a out-of-range block is an indirect block, do not attempt to dump its contents.

Output Files

For each different kind of entry, output its summary in a separate file. The names of your files should be:

- super.csv ...for super block;
- group.csv ...for group descriptors;
- bitmap.csv ...for free inodes and blocks;
- inode.csv ...for inodes;
- directory.csv ...for directory entries; and
- indirect.csv ...for indirect block entries.

In these csv files, each line represents one entry, and should be ended with a single new-line character ('\n', 0x0a). The field orders in those csv files should be the same as listed above, and your program should **use and only use** a comma to separate each field.

For your convenience, here are the exact six csv files generated by our solution: super.csv, group.csv, bitmap.csv,inode.csv, directory.csv, and indirect.csv.

SUBMISSION:

Please carefully check the Rubric section (especially the **Code review** subsection) and **all (latest) footnotes** before submitting your code.

Your tarball should have a name of the form lab3a-*studentID*.tar.gz and should be submitted via CCLE.

We will test your work on a SEASnet GNU/Linux server running RHEL 7 (this is on lnxsrv09). You would be well advised to test your submission on that platform before submitting it.

RUBRIC:

Value	Feature
	Packaging and build (10%)
3%	untars expected contents
3%	clean build w/default action (no warnings)
2%	Makefile has clean and dist targets
2%	reasonableness of README contents

Code review (15%)

You can check header files like ext2_fs.h to have an idea about the variable types. HOWEVER, you shall write up your own code for the calculations like block/inode (number), indirect block pointer, various different kinds of offset, and so on. So in sum, do all the calculations with your own code, do not use macros/functions provided in header files like ext2fs.h.

5% overall readability and reasonableness, correct program name and csv filenames
10% image file investigation correct (use pread() and write your own code for calculations)

Results (75%)

⅔ points shall go to outputting the required fields correctly, and ⅓ points go to correct lines (penalty will be given for extra/missing lines).

10% super.csv
10% group.csv
10% bitmap.csv
15% inode.csv
15% directory.csv
15% indirect.csv

---

[1] We only recognize a (small) subset of the file types: 'f' for regular file, 'd' for directory, and 's' for symbolic links. For other types, use '?'.

[2] Double-quoted. Don't worry about how to print out non-graphical characters or embedded quotes.