File System Interpretation

INTRODUCTION:

We are all familiar with the characteristics of the files and directories in which we store all of our data. As with many other persistent objects, their functionality, generality, performance and robustness all derive from the underlying data structures used to implement them. In this project we will design and implement a program to read the on-disk representation of a file system, analyze it, and summarize its contents. In the next project, we will write a program to analyze this summary for evidence of corruption.

This project can be broken into two major steps:

- Understand the on-disk data format of the EXT2 file system. We will mount a provided image file on your own Linux system and explore it with familiar file navigation commands and *debugfs(8)*.
- Write a program to analyze the file system in that image file and output a summary to standard out (describing the super block, groups, free-lists, inodes, indirect blocks, and directories).

The second part may involve much more code than we have written in previous projects, but the first part is (by far) the more difficult. Once the underlying data structures are understood, the actual code is likely to be fairly simple.

RELATION TO READING AND LECTURES:

This project more deeply explores the filesystem structures described in Arpaci chapter 39.
The images we will be working with are EXT2 file systems, as described in sections 40.2-40.5.

The lectures on file systems, file system performance, and reliability describe why data structures like those in this project are used to manage file systems.

PROJECT OBJECTIVES:

- Primary: reinforce the basic file system concepts of directory objects, file objects, and free space.
- Primary: reinforce the implemenation descriptions provided in the text and lectures.
- Primary: gain experience researching, examining, interpreting and processing information in complex binary data structures.
- Primary: gain experience with examining interpreted and raw hex dumps of complex data structures as a means of developing an understanding of those data structures.
- Secondary: gain practical experience with typical on-disk file system data formats.

DELIVERABLES:

A single tarball (`.tar.gz`) containing:

- (at least) one C/C++ source module that compiles cleanly with no errors or warnings).
- A `Makefile` to build and run the deliverable program. The higher level targets should be:

    - default ... compile your program (with the **`-Wall`** and **`-Wextra`** options) to produce an executable named `lab3a`
    - **dist** ... create the deliverable tarball
    - **clean** ... delete all programs and output generated by the `Makefile`.

- a `README` text file containing descriptions of each of the included files and any other information about your submission that you would like to bring to our attention (e.g., research, limitations, features, testing methodology).

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 privacy and integrity, file system 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.

PREPARATION

To perform this assignment, you may need to study a few things:

- debugfs(8), a tool for exploring on-disk file system structures.
- pread(2), an alternative to *read(2)* for random-access file processing.
- a comprehensive overview of the EXT2 file system format.
- a slightly simplified version of the Linux header file that defines the format of the EXT2 file system. Do not assume that the standard header file will be available on the test system. Please use this header file and include it in your submission. You cannot modify this header file, but you are free to add new files of your own.

PART 1: Exploring an EXT2 Image

In order to mount and explore a file system (even one stored in an ordinary file) you will need the ability to run privileged commands (e.g., *mount(8)*). Since you will not have *sudo(8)* access on departmental servers, you will have to do this exploration on your own personal Linux system.

Download this (1-2MB) file system image, and mount it (read only) onto your own Linux system, with the following commands:

```
mkdir fs
sudo mount -o ro,loop EXT2_test.img fs
```

The `loop` option means that the file system image is stored in a file rather than on a device. The `ro` option means **read only**, to prevent you from accidentally changing the image.

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/C++ program to interpret the diskimage file, you can explore it further using *debugfs(8)* (on your own Linux system). You will, in the process of writing your code, surely encounter many questions about how to interpret the values in various fields. Reading the specifications is seldom enough to enable us to fully understand complex data structures. Welcome to the real world! Learning how to complement research with examination and experimentation to understand a complex system is a skill that you are expected to develop and demonstrate in this project. The supplied images and exploration tools can be used to examine real instances of super blocks, group summaries, I-nodes, directories, etc.

Some particularly helpful *debugfs(8)* commands are: `stats`, `stat`, `bd`, `testi`, and `testb`. While *debugfs* can interpret data structures for you, you may find that a simple hex dump of the associated block provides you with more detailed information.

Warnings

If you mount the `trivial.img` image read/write, even read commands (like *ls(1)*) will cause changes to I-node access times. If you want to be able to compare your analysis with the golden `trivial.csv` output we have provided, you must work from an unmodified version of `trivial.img`.

To ensure you are correctly interpreting the file system image, we have included many unusual things, which might not be properly handled by a naive implementation:

- sparse files with large unallocated areas between allocated blocks
- very 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 or non-ASCII names
- directories that span multiple blocks, go beyond the direct blocks, and have obsolete entries for deleted files

All of these are completely legal and do not represent any sort of corruption.

PART 2: Summarizing an EXT2 Image

In this step, you will write a C/C++ program called `lab3a` that:

- Reads a file system image, whose name is specified as a command line argument. For example, we may run your program with the above file system image using the a command like:

```
./lab3a EXT2_test.img
```

- Analyzes the provided file system image and produces (to standard out) CSV summaries of what it finds. The contents of these CSV lines described below. Your program must output these files with

exactly the same formats as shown below. We will use *sort(1)* and *diff(1)* to compare your csv output with ours, so the order of output lines does not matter, but a different format (even extra white space) will make your program fail the test.

Please note that, even if you cannot mount the provided image file and run *debugfs* on departmental servers, your lab3a program should, like previous assignments, be able to run on departmental servers.

There are six types of output lines that your program should produce, 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. We have also included (for testing purposes) a much smaller image as well as a correct output summary.

You are free to produce additional commentary to stderr, but only file system summary information should be logged to stdout.

superblock summary

A single new-line terminated line, comprised of eight comma-separated fields (with no white-space), summarizing the key file system parameters:

1. `SUPERBLOCK`
2. total number of blocks (decimal)
3. total number of i-nodes (decimal)
4. block size (in bytes, decimal)
5. i-node size (in bytes, decimal) Depends on revision number
6. blocks per group (decimal)
7. i-nodes per group (decimal)
8. first non-reserved i-node (decimal) Depends on revision number

group summary

Scan each of the groups in the file system. For each group, produce a new-line terminated line for each group, each comprised of nine comma-separated fields (with no white space), summarizing its contents.

1. ~~`GROUP`~~
2. ~~group number~~ (decimal, starting from zero) Depends on revision number
3. ~~total number of blocks in this group~~ (decimal) Explicit calculation needed
4. ~~total number of i-nodes in this group~~ (decimal) Explicit calculation needed
5. ~~number of free blocks~~ (decimal)
6. ~~number of free i-nodes~~ (decimal)
7. ~~block number of free block bitmap for this group~~ (decimal)
8. ~~block number of free i-node bitmap for this group~~ (decimal)
9. ~~block number of first block of i-nodes in this group~~ (decimal) Could this use results from SB Summary 8?

Note that most Berkeley-derived file systms (like EXT2) support both blocks and fragments, which may have different sizes. The block is the preferred unit of allocation. But in some cases, fragments may be used (to reduce internal fragmentation loss). Block addresses and the free block list entries are based on the fragment size, rather than the block size. But, in the images we give you, the block and fragment sizes will be the same.

One of the major features included EXT2 file systems is support for multiple cylinder groups:

- all cylinder groups but the last have the same number of blocks and I-nodes; the last has the residue (e.g., blocks/fs modulo blocks/group).
- each group, in addition to its group summary, also (for redundancy) starts with a copy of the file system superblock.

But, in the images we give you, there will be only a single group.

free block entries

Scan the free block bitmap for each group. For each free block, produce a new-line terminated line, with two comma-separated fields (with no white space).

1. `BFREE`
2. number of the free block (decimal)

Take care to verify that you:

1. understand whether 1 means allocated or free.
2. have correctly understood the block number to which the first bit corresponds.
3. know how many blocks are in each group, and do not interpret more bits than there are blocks in the group.

free I-node entries

Scan the free I-node bitmap for each group. For each free I-node, produce a new-line terminated line, with two comma-separated fields (with no white space).

1. `IFREE`
2. number of the free I-node (decimal)

Take care to verify that you:

1. understand whether 1 means allocated or free.
2. have correctly understood the I-node number to which the first bit corresponds.
3. know how many I-nodes are in each group, and do not interpret more bits than there are I-nodes in the group.

I-node summary

Scan the I-nodes for each group. For each allocated (non-zero mode and non-zero link count) I-node, produce a new-line terminated line, with up to 27 comma-separated fields (with no white space). The first twelve fields are i-node attributes:

1. `INODE`
2. inode number (decimal)
3. file type ('f' for file, 'd' for directory, 's' for symbolic link, '?" for anything else)
4. mode (low order 12-bits, octal ... suggested format "%o")
5. owner (decimal)
6. group (decimal)
7. link count (decimal)

8. time of last I-node change (mm/dd/yy hh:mm:ss, GMT)
9. modification time (mm/dd/yy hh:mm:ss, GMT)
10. time of last access (mm/dd/yy hh:mm:ss, GMT)
11. file size (decimal)
12. number of (512 byte) blocks of disk space (decimal) taken up by this file

The *number of blocks* (field 12) should contain the same value as the *i_blocks* field of the I-node. There are a few interesting and non-obvious things about this number:

1. This number is in units of 512 byte blocks, even if the file system block size is something else (e.g. 1024 or 4096 byte blocks).
2. This number (times 512) may be smaller than the file size, as it includes only blocks that have actually been allocated to the file. A very large file might be sparse, in that some parts of the file may not have actually been written, and take up no disk space, but will read back as zeroes.
3. This number (times 512) may be larger than the file size because it includes not only data blocks, but (single, double, and tripple) indirect blocks that point to data blocks.

For ordiary files (type 'f') and directories (type 'd') the next fifteen fields are block addresses (decimal, 12 direct, one indirect, one double indirect, one triple indirect).

Symbolic links may be a little more complicated. If the file length is less than or equal to the size of the block pointers (60 bytes) the file will contain zero data blocks, and the name is stored in the space normally occupied by the block pointers. If this is the case, the fifteen block pointers should not be printed. If, however, the file length is greater than 60 bytes, print out the fifteen block nunmbes as for ordinary files and directories.

directory entries

For each directory I-node, scan every data block. For each valid (non-zero I-node number) directory entry, produce a new-line terminated line, with seven comma-separated fields (no white space).

1. `DIRENT`
2. parent inode number (decimal) ... the I-node number of the directory that contains this entry
3. logical byte offset (decimal) of this entry within the directory
4. inode number of the referenced file (decimal)
5. entry length (decimal)
6. name length (decimal)
7. name (string, surrounded by single-quotes). Don't worry about escaping, we promise there will be no single-quotes or commas in any of the file names.

indirect block references

The I-node summary contains a list of all 12 blocks, and the primary single, double, and triple indirect blocks. We also need to know about the blocks that are pointed to by those indirect blocks. For each file or directory I-node, scan the single indirect blocks and (recursively) the double and triple indirect blocks. For each non-zero block pointer you find, produce a new-line terminated line with six comma-separated fields (no white space).

1. `INDIRECT`
2. I-node number of the owning file (decimal)

3. (decimal) level of indirection for the block being scanned ... 1 for single indirect, 2 for double indirect, 3 for triple
4. logical block offset (decimal) represented by the referenced block. If the referenced block is a data block, this is the logical block offset of that block within the file. If the referenced block is a single- or double-indirect block, this is the same as the logical offset of the first data block to which it refers.
5. block number of the (1, 2, 3) indirect block being scanned (decimal) . . . not the highest level block (in the recursive scan), but the lower level block that contains the block reference reported by this entry.
6. block number of the referenced block (decimal)

Logical block is a commonly used term. It ignores physical file structure (where data is actulally stored, indirect blocks, sparseness, etc) and views the data in the file as a (logical) stream of bytes. If the block size was 1K (1024 bytes):

- bytes 0-1023 would be in logical block 0
- bytes 1024-2047 would be in logical block 1
- bytes 2048-3071 would be in logical block 2
  ...

You can confirm your understanding of logical block numbers by looking at the `INDIRECT` entries in the sample output.

If an I-node contains a triple indirect block:

- the triple indirect block number would be included in the INODE summary.
- `INDIRECT` entries (with level 3) would be produced for each double indirect block pointed to by that triple indirect block.
- `INDIRECT` entries (with level 2) would be produced for each indirect block pointed to by one of those double indirect blocks.
- `INDIRECT` entries (with level 1) would be produced for each data block pointed to by one of those indirect blocks.

Sample Output

We have provided a very simple test file system image as well as a correct output summary that you can download and test with. Your program should be able to generate (modulo line ordering) the same output. The grading program will run your program on a variety of other file system images, and check whether or not your output is identical to the golden output. Any differences (even white space or a case error) will result in a loss of all points for that test.

SUMMARY OF EXIT CODES:

- 0 . . . analysis successful
- 1 . . . bad arguments
- 2 . . . corruption detected or other processing errors

SUBMISSION:

Your **README** file must include lines of the form:

NAME: *your (comma separated) name(s)*
EMAIL: *your (comma separated) email(s)*
ID: *your (comma separated) student ID(s)*

Your name, student ID, and email address should also appear as comments at the top of your `Makefile` and each source file. Your ID should be in the XXXXXXXXX format, not the XXX-XXX-XXX format. If this is a team submission, the names, e-mail addresses, and student IDs should be comma-separated.

Your tarball should have a name of the form `lab3a-`*studentID*`.tar.gz`. If you are doing this project as a team submission, either of your student ID numbers can be used.

You can sanity check your submission with this test script. There will be no manual re-grading on this project. Submissions that do not pass the test script are likely to receive very low scores. Note that the sanity check runs are timed. If your program is implemented inefficiently (e.g. too many scans, using too much memory) it may run so slowly as to time out and fail. If this happens to you, examine your approach and look for a more efficient way to obtain the required information. Also note that passing this sanity check does not guarantee a 100% score on the project. You are responsible for testing your own code, and the sanity check script is merely one tool for testing, not a guarantee that everything is correct.

We will test it on a departmental Linux server. You would be well advised to test your submission on that platform before submitting it.

You may add files not specified in this page into the tarball for your submission, if you feel they are helpful. If you do so, be sure to mention each such file by name in your README file. Also be sure they are properly handled during the dist and clean operations in your Makefile.

GRADING:

Points for this project will be awarded:

value	feature
	Packaging and build (10% total)
3%	un-tars expected contents
3%	clean build with default action (no warnings)
2%	correct `clean` and `dist` targets
2%	reasonableness of `README` contents
	Results (85% total)
10%	superblock summary
10%	group summaries
10%	free block entries
10%	free I-node entries
15%	I-node summaries
15%	directory entries
15%	block references
	Code Review (5%)
5%	general organization and readability