File System Consistency Analysis

INTRODUCTION:

A file system is an example of a large aggregation of complex and highly inter-related data structures. Such systems are often at risk for corruption, most commonly resulting from incomplete execution of what should have been all-or-none updates. Unless we can so cleverly design our data structures and algorithms as to render such errors impossible, it is usually necessary to articulate consistency assertions and develop a utility to audit those systems for consistency, and (where possible) repair any detected anomalies. In part A of this project we wrote a program to examine on-disk data structures and summarize their contents. In this second part, we will analyze those summaries to detect errors and inconsistencies.

All previous projects have been implemented in C, which exposes the underlying system calls and is well suited for low level operations on well-defined data structures. This project involves trivial text processing (to read .csv input) and the assembly and processing of an internal state model. While this can certainly be done in C/C++, you may find it much easier to do in a higher level language (e.g. Python). You are free implement this project in any language that is supported on the departmental servers (where we will do the testing).

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.
This project goes much deeper than the introductory integrity discussion in sections 42.1-2.

This project involves many of the same concepts (superblocks, I-nodes, free lists, etc) you dealt with in the previous project, and the input to the analysis program in this project is the same (csv summary) format as the output produced in the previous project. But, in this project, all analysis will be of csv file system summaries. You will not be scanning raw file systems, and you will not need any of your code from the previous project.

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 examining, interpreting and processing information in complex binary data structures.
- Primary: reinforce the notions of consistency and integrity and apply them to a concrete and non-trivial problem.

DELIVERABLES:

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

- (at least) one source module that builds/executes cleanly with no errors or warnings.

- A `Makefile` to build and run the deliverable program. The higher level targets should be:
  - default ... build your program to produce an executable named `lab3b`

    - **dist** ... create the deliverable tarball
    - **clean** ... delete all programs and output generated by the `Makefile`, and return the directory to its freshly un-tar-ed state.

- 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).

  NOTE: do not create or edit your `README` file on a Windows system, as the gratuitious carriage-returns may cause processing errors in the auto-grading scripts.

PROJECT DESCRIPTION:

Write a program to analyze a file system summary (a `.csv` file in the same format produced for the previous file system project) and report on all discovered inconsistencies. Detected inconsistencies should be reported to standard out. Execution errors (e.g., invalid arguments or unable to open required files) should be reported to standard error.

Your executable should be called `lab3b` and accept a single, required, command line argument, the name of the file to be analyzed. If your language does not require pre-compilation (e.g., Python) create a default rule in your `Makefile` that merely creates a `lab3b` link, or prints a success message. If your language does not allow a program to have such a name, include a shell script (named `lab3b`) that will run your program.

The results grading for this project will be entirely automated, and messages produced in an incorrect format will receive no points. We are providing a sanity check script that will do a simple validation of your package, and confirm correct output for a two-digit number of basic errors. For a simple clean summary, you can use the correct output sample we provided for the previous project. If you want a set of corrupted summaries to test with, you can pull down copies of the summaries (with names like `P3B-test_1.csv`) and corresponding correct output (with names like `P3B-test_1.err`) used by the sanity check script.

Part of your score will be based on your ability to correctly recognize and report on the problems in the supplied file system summaries, but your program will also be tested on several other file systems with a wider range of anomalies. You can lose points for mis-reporting errors, failing to report errors, or for reporting errors that are not present.

This is a summary of the errors your program should check for, and a sample error message for each:

Block Consistency Audits

Every block pointer in an I-node or indirect block should be valid (a legal data block, within the file system) or zero. Examine every single block pointer in every single I-node, direct block, indirect block, double-indirect block, and triple indirect block to ascertain that this is true. But, remember that for symbolic links with a short length (60 bytes or less) the i-node block pointer fields do not contain block numbers, and so should not be analyzed.

Look at all block pointers in the I-node, not merely those within the indicated file size. If any block pointer is not valid (a legal data block within the file system), an error message like one of the following (depending on precisely the nature of the incorrect pointer that is found) should be generated to stdout:

```
INVALID BLOCK 101 IN INODE 13 AT OFFSET 0
```

```
        INVALID INDIRECT BLOCK 101 IN INODE 13 AT OFFSET 12
        INVALID DOUBLE INDIRECT BLOCK 101 IN INODE 13 AT OFFSET 268
        INVALID TRIPLE INDIRECT BLOCK 101 IN INODE 13 AT OFFSET 65804
        RESERVED INDIRECT BLOCK 3 IN INODE 13 AT OFFSET 12
        RESERVED DOUBLE INDIRECT BLOCK 3 IN INODE 13 AT OFFSET 268
        RESERVED TRIPLE INDIRECT BLOCK 3 IN INODE 13 AT OFFSET 65804
        RESERVED BLOCK 3 IN INODE 13 AT OFFSET 0
```

An `INVALID` block is one whose number is less than zero or greater than the highest block in the file system.
A `RESERVED` block is one that could not legally be allocated to any file because it should be reserved for file system metadata (e.g., superblock, cylinder group summary, free block list, ...).
The logical `OFFSET` values are as described for the previous project.

Note that the reported offsets should be block numbers (byte offsets divided by 1024). The offset associated with an indirect blocks should be that associated with the first data block it points to (as in the previous project).

Every legal data block (every block between the end of the I-nodes and the start of the next group) should appear on on the free block list, or be allocated to exactly one file. Examine the free list to determine whether or not this is the case. If a block is not referenced by any file and is not on the free list, a message like the following should be generated to stdout:

```
        UNREFERENCED BLOCK 37
```

A block that is allocated to some file might also appear on the free list. In this case a message like the following should be generated to stdout:

```
        ALLOCATED BLOCK 8 ON FREELIST
```

If a legal block is referenced by multiple files (or even multiple times in a single file), messages like the following (depending on precisely where the references are) should be generated to stdout for each reference to that block:

```
        DUPLICATE BLOCK 8 IN INODE 13 AT OFFSET 0
        DUPLICATE INDIRECT BLOCK 8 IN INODE 13 AT OFFSET 12
        DUPLICATE DOUBLE INDIRECT BLOCK 8 IN INODE 13 AT OFFSET 268
        DUPLICATE TRIPLE INDIRECT BLOCK 8 IN INODE 13 AT OFFSET 65804
```

Note that you will not know that a block is multiply referenced until you find the second reference. You will have to figure out a way to go back and report ALL of the references. We must report all of the references, because each of them is likely to have been corrupted.

I-node Allocation Audits

We can tell whether or not an I-node is allocated by looking at its type and presence in the CSV file. An allocated I-node will have some valid type (e.g., file or directory). Unallocated I-nodes (whose type should be zero) should not appear in INODE summaries, but should be in the free list. Scan through all of the I-nodes to determine which are valid/allocated. Every unallocated I-Node should be on a free I-node list. Compare your list of allocated/unallocated I-nodes with the free I-node bitmaps, and for each discovered inconsistency, a message like one of the following should be generated to stdout:

```
        ALLOCATED INODE 2 ON FREELIST
        UNALLOCATED INODE 17 NOT ON FREELIST
```

Directory Consistency Audits

Every allocated I-node should be referred to by the number of directory entries that is equal to the reference count recorded in the I-node. Scan all of the directories to enumerate all links.

For any allocated I-node whose reference count does not match the number of discovered links, an error message like the following should be generated to stdout:

```
        INODE 2 HAS 4 LINKS BUT LINKCOUNT IS 5
```

This message should also be used to report unreferenced I-nodes:

```
        INODE 17 HAS 0 LINKS BUT LINKCOUNT IS 1
```

Directory entries should only refer to valid and allocated I-nodes. An `INVALID` I-node is one whose number is less than 1 or greater than the last I-node in the system. While scanning the directory entries, check the validity and allocation status of each referenced I-node. For any reference to an invalid or unallocated I-node, an error message like the following should be generated to stdout:

```
        DIRECTORY INODE 2 NAME 'nullEntry' UNALLOCATED INODE 17
        DIRECTORY INODE 2 NAME 'bogusEntry' INVALID INODE 26
```

We also know that every directory should begin with two links, one to itself (**.**) and one to its parent (**..**). While scanning each directory, check for the correctness of these two links and, for each detected inconsistency, a message like one of the following should be generated to stdout:

```
        DIRECTORY INODE 2 NAME '..' LINK TO INODE 11 SHOULD BE 2
        DIRECTORY INODE 11 NAME '.' LINK TO INODE 2 SHOULD BE 11
```

NOTE

As in the previous project, the file system summaries we use to test your submission will describe file systems with only a single group.

SAMPLE DAMAGED FILE SYSTEM SUMMARIES AND OUTPUT

The sanity check script will automatically download a large number of damaged file system summaries. Run your program against them, and compare your output with (what I believe to be) the correct error reports. Note that the sanity check (and grading) runs are timed. If your program is implemented inefficiently (e.g. too many scans of the file, or 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 do the required analysis.

As with the previous project, your output will be sorted before it is compared with the golden results, so the order in which you output errors is unimportant.

SUMMARY OF EXIT CODES:

- 0 ... successful execution, no inconsistencies found.
- 1 ... unsuccessful execution, bad parameters or system call failure.
- 2 ... successful execution, inconsistencies found.

SUBMISSION:

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

> **NAME:** *your (comma separted) name(s)*
> **EMAIL:** *your (comma separted) email address(es)*
> **ID:** *your (comma separted) 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 `lab3b-`*studentID*`.tar.gz`. If this is a team submission, only one student ID need appear in the tarball's name.
You can sanity check your submission with this test script.
Projects that do not pass the test script are likely to receive very poor scores! Note, however, 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.

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.

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

GRADING:

Points for this project will be awarded:

value	**feature**
	Packaging and build (10% total)
3%	un-tars expected contents
3%	clean build
2%	correct `clean` and `dist` targets
2%	reasonableness of `README` contents
	Results (80% total)
10%	illegal block pointer detection and reporting
10%	reserved block pointer detection and reporting

5%	unallocated block detection and reporting
15%	duplicately referenced block detection and reporting
5%	I-node allocation and free list error detection and reporting
10%	incorrect link count detection and reporting
5%	unreferenced I-node detection and reporting
5%	invalid I-node in directory entry detection and reporting
5%	unallocated I-node in directory entry detection and reporting
10%	., .. directory entries error detection and reporting
	Code Review (10%)
5%	intelligent choice of language and exploitation of its features
5%	general organization and readability