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.

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, you may find it 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.

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 ... compile your program to produce an executable named `lab3a`
  - **dist** ... create the deliverable tarball
  - **clean** ... delete all generated programs and output, returning 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, use of slip days).

PROJECT DESCRIPTION:

Write a program to analyze a file system summary 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 (at least) a single, required, command line argument, the name of the file to be analyzed. If your language does not allow a program to have such a name, include a shell script (named `lab3b`) that will run 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). Examine every single block pointer in every single I-node, direct block, indirect block, double-indirect block, and tripple indirect block to ascertain that this is true. 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 incorrect pointer 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 TRIPPLE 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 TRIPPLE INDIRECT BLOCK 3 IN INODE 13 AT OFFSET 65804
RESERVED BLOCK 3 IN INODE 13 AT OFFSET 0
```

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 does not appear in any file or 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 block is referenced by multiple files, messages like the following (depending on precisely where the references are) should be generated to stdout <u>for each reference to that block</u>:

```
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 TRIPPLE 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.

I-node Allocation Audits

We can tell whether or not an I-node is allocated by looking at its type. An unallocated I-node should have a type of zero, and an allocated I-node should have some valid type (e.g. file or directory). Scan through all of the I-nodes to determine which are valid/allocated. Every unallocated 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 a 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
```

Directory entries should only refer to valid and allocated I-nodes. 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 validity 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
```

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.

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 name*
> **EMAIL:** *your email*
> **ID:** *your student ID*

Your name, student ID, and email address should also appear as comments at the top of your `Makefile` and each source file.

Your tarball should have a name of the form `lab3b-`*studentID*`.tar.gz`.
You can sanity check your submission with this [test script](#).
Projects that do not pass the test script will not be accepted!

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%	., .. error detection and reporting
	Code Review (10%)
5%	intelligent choice of language and exploitation of its features

5%   general organization and readability