Mohammed Junaid Ahmed (104-293-008)

Robert Abbott (004-432-123)

**Lab 3 Design Problem: File System Fixer**

**Feature Specification:**

An important feature in a real file system is the ability to recover from and repair minor errors that occur when the system is shut down improperly, hardware fails, or the CPU is halted before the cache is synced. Ideally this file system fixing utility would use its knowledge of the file system’s structure to perform some basic consistency checks and either repair inconsistencies or inform the user that the issue exists.

Given a knowledge of the structure of ospfs there are some basic, but valuable tests the file system checker/fixer will be able to perform on the file system:

**Superblock sanity check:**

The first logical place to test is the superblock. The superblock contains metadata on the entire file system, and a corrupted superblock can be a serious issue. In the UNIX file system there are backup superblocks scattered throughout the disk to ensure the structure of the file system is not unrecoverably lost. However a basic sanity check will still be run using data from the superblock. The FS fixer will check that the number of blocks and inodes that have been allocated do not surpass the FS size specified in the superblock. If this condition is violated the program will return with a message indicating the FS has been corrupted.

**Block validity:**

Blocks are containers that hold the files stored in a file system. An important test that will be performed is to check the validity of all the blocks in the system. This means that if a block is marked as used some inode claims it, and if a block is marked not used, no inode claims it. Free blocks must also be initialized to zero. If issues are found at this stage, they can be repaired by the fixer.

**Bitmap integrity:**

The bitmap maps bits to blocks and the bit indicates whether or not the block is being used. Once the blocks have been checked for validity (and fixed if necessary), the next step is to check the bitmap. At this stage the blocks will be marked correctly. The bitmap will be checked against the blocks and fixed if there are any discrepancies.

**Dangling inodes:**

It is possible for files to be deleted from the file system, but their inodes and associated blocks are not. The fixer can find these dangling inodes by traversing the directory structure. If the fixer finds inodes that do not have a corresponding file in the file system it will place the inode in the lost and found and the user will decide what to do with the inode.

Finally there is a question of what information should be returned to the user once the file system fixer has run. At the end of execution the FS fixer will return a list of the issues it found with an indication of whether or not that issue was fixed.

These specifications constitute a file system checker/fixer facility that will be able to detect errors in the ospfs file system. For some issues the fixer can detect it will be able to take corrective action. Upon termination of execution the fixer will inform the user what errors were found and what actions were taken.

**Implementation:**

The first step to implementing the file system checker (fsck) is to turn the image file into a useful, readable format. This is done by first reading the superblock from the image file, then using that information to determine the number of inodes, number of blocks, and first block. Then the remainder of the file system is read into a buffer so that the entire file system is stored in the fsck programs memory. The code below reads the entire file system into memory:

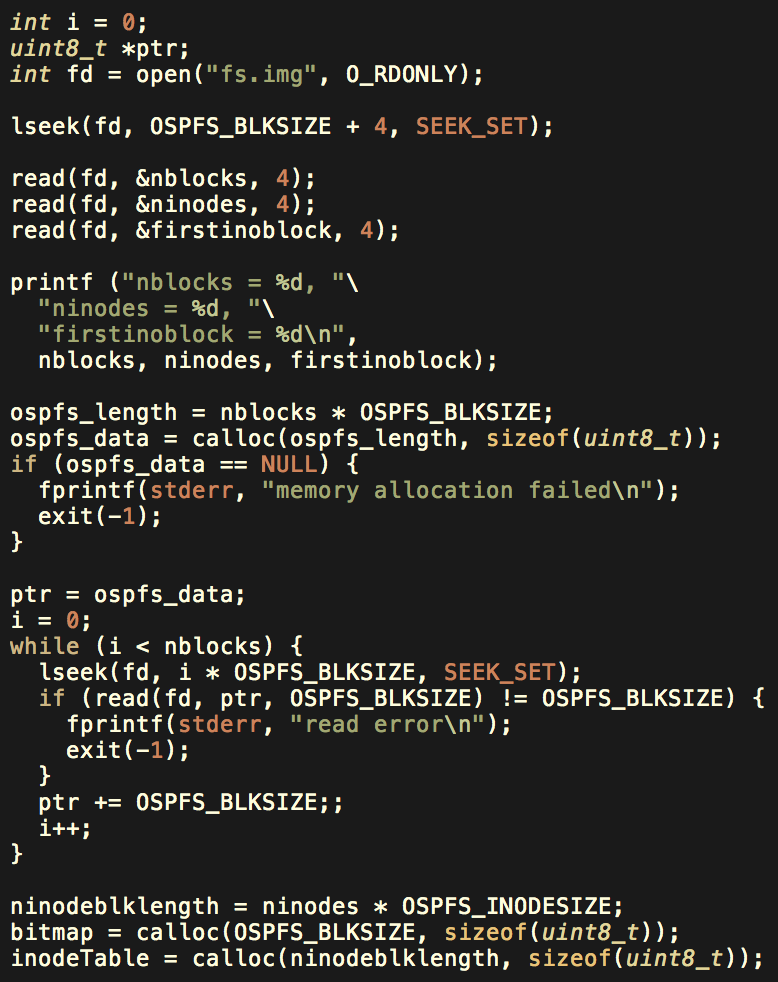


figure 1: read file system into memory

The above code reads the file system into memory then allocates space for the bitmap, inodes and inode table. These structures are later populated and used to run various tests on the file system.

Once the file system has been read into memory, the first step is to do some basic sanity checks on the superblock. By reading the first four bytes of the superblock the magic number is found which ensures that the correct file system is being checked. The next four bytes indicate the number of blocks in the file system. Using lstat the size of the file system is verified to be equal to the size indicated by the superblock. If these sizes do not match fsck returns and indicates that the superblock is corrupted, in this case no corrective action can be taken. The next step in verifying the superblock is checking that the blocks reserved for file system metadata (1,2 and some portion of block 3) are marked as used. If they are marked unused in the bitmap, this will be corrected by setting those bits to be marked in the bitmap. The following code performs these tests. First it checks that the file system size it correct. Next it checks that the first 3 blocks are initialized properly:

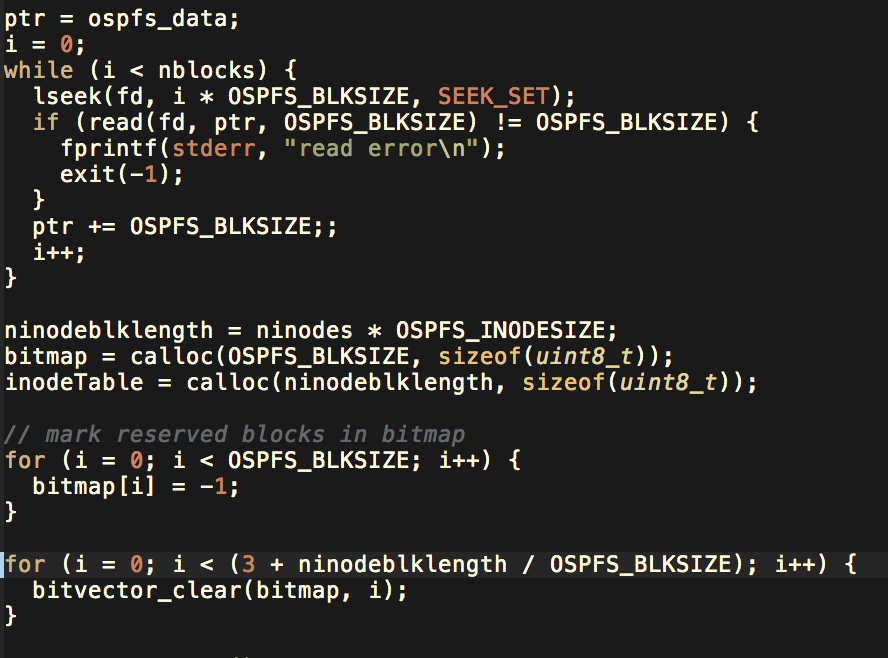


Figure 2: superblock checking

If the superblock is correct or can be corrected the program will move to the next set of tests. The first test is to eliminate dangling inodes and ensure that only valid inodes remain in the system. This is accomplished by going through all the direntries and making sure the inode they are associated with is marked used. If not, the direntry is deleted. Any inode that is marked as used but not associated with a direntry will also be deleted during this pass. The following code performs the check for dangling inodes. It uses the ospfs\_inode method from lab3 to get nodes by their inode number. Next the program checks that if an inode has a link count greater than zero the inode is claimed by some direntry. If this is not the case the node is put in lost and found.

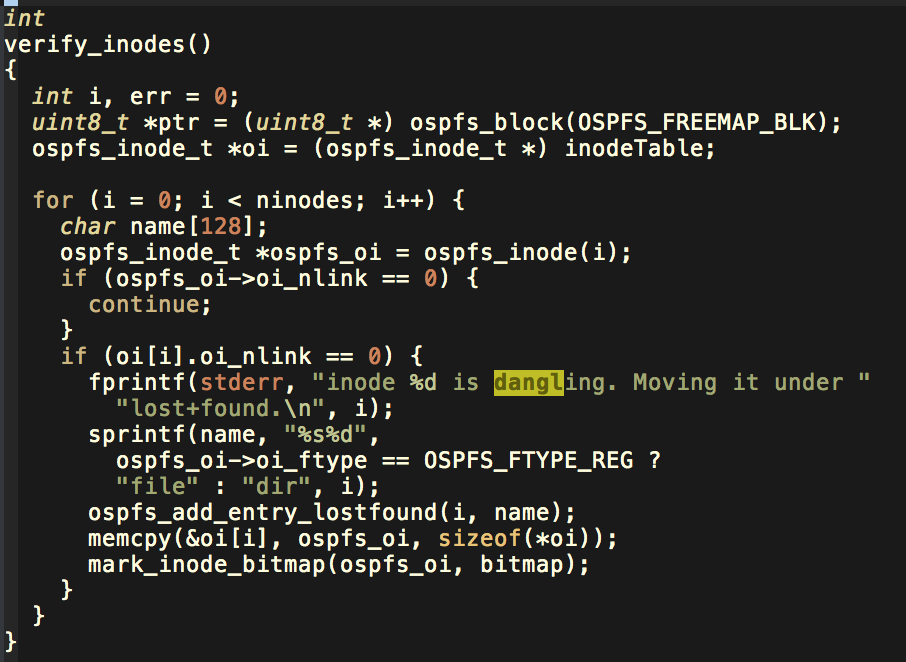


figure 3: finding dangling inodes

The next tests involve verifying the blocks that make up the system then checking both the inodes and bitmap with the blocks. The program first goes through the inodes and determines which blocks are claimed by inodes. The program then goes through the blocks and fixes any incorrectly marked blocks. The program has the entire file system in memory and the size and number of blocks and inodes is known. This means getting inode and block information simply requires looking at the correct index of the array that contains the file system. Once the program goes through the inodes and determines which blocks should be marked as used (by storing their numbers in an array), the program checks all the blocks in the system and fixes them if they are incorrectly marked. In this case the program trusts that the inodes correctly claimed the blocks. The program prints out any changes in makes so the user can track what the program has done.

The next step is checking the bitmap with the corrected blocks. This is accomplished by making another pass through the blocks and verifying that their status is consistent with what is listed in the bitmap. If the bitmap does not match a block the bitmap will be corrected to match what is in the blocks.

Error reporting is accomplished during the program’s execution. Rather log all the information or ask the user to decide how to handle a bug, the program prints out errors it finds and changes it makes as it goes. This creates a log of sorts that allows the user to see that the program is repairing the file system properly.

The whole program can be run by calling the main function. The main function initializes the fsck by reading the file system into memory. The main then performs a check and fix function which performs all the tests by calling their respective helper methods. Once the tests have run the main calls commit.

**Testing:**

In order to test the fsck program, the file system had to first be corrupted. This was accomplished using a shell script which would corrupt the file system then run the fsck program and verify that the output was consistent. The shell script corrupts the file system in two ways. First the shell script corrupts the bitmap by changing one of the bits in the map. Second, the shell corrupts the inode block by writing a zero to of the inodes. The fsck program is able to detect and correct both of these types of corruption. The shell script used to corrupt and test the code is shown below:

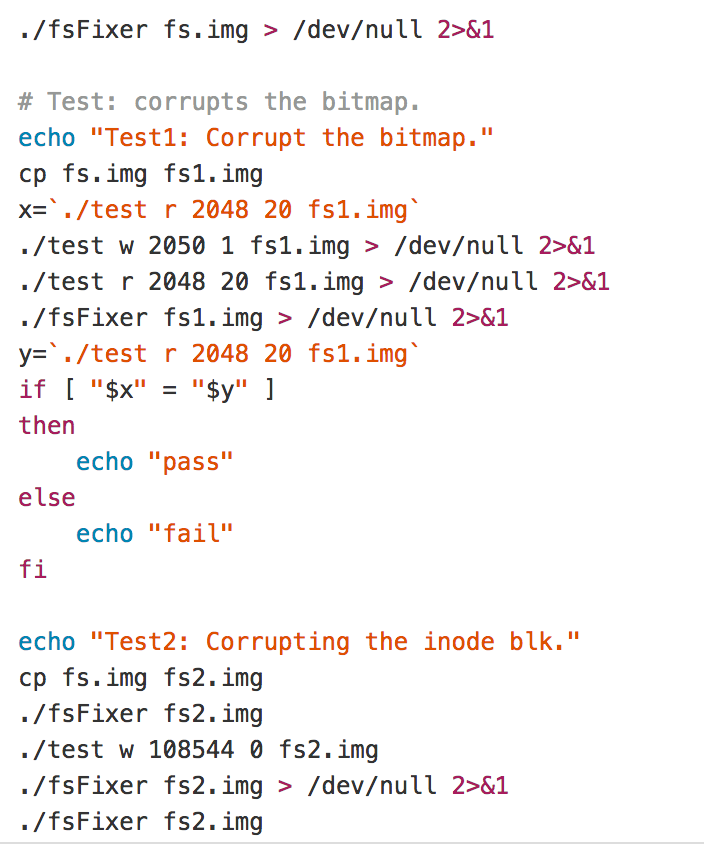


figure 4: test script

**Results:**

This implementation of fsck for the ospfs file system is able to perform the tests described in the specification. However there are some small components a normal fsck would have that are not implemented. These include getting user input on how to handle certain bugs, and handling of inodes placed in the ‘lost and found’. In building this file system checker we learned a lot about how fsck works in a unix system which helped inform our design decisions on our own file system checker. Lab 3 gave us a very good understanding of the structure of the ospfs file system which made implementing the fsck much easier. Once the file system was read into memory, checking the various components was relatively straightforward.