Project Overview

Motivation

The overall goal of our project is to create a file system that is more resilient to corruption. We hoped to create a system that is able to determine when a file or directory has been corrupted, and to recover from those corruptions in certain cases. We drew inspiration from ZFS to develop a system that is able to both warn the user of corruptions and recover from those disk corruptions in certain cases. Another major consideration was the tree structure of the file system. If a directory node is corrupted, the files and directories below it also not be accessible through the traditional directory structure. If we can protect nodes that have many children, we are able to maintain the integrity of a higher proportion of the file system.

Overview

The major concept that we borrowed from ZFS is that of *ditto blocks*. A ditto block is a mirror of a file's inode that points to a new set of data blocks that duplicate the original file's content exactly. In our system, each inode has either 0, 1, or 2 ditto blocks. We added child1 and child2 fields to the inode which point to the ditto blocks for a given file.

To determine if a file is corrupted, we implemented a checksum function that takes an inode as an argument. The checksum function iterates over every data block in a file and computes a value that is then stored in the inode and dinode. Then, to determine if a file is corrupted, the checksum is recomputed then compared against the stored value. When attempting to access a file, the ilock operation performs this check. If a file is determined to be corrupted, we attempt to find an uncorrupted version in the ditto blocks. If we find an uncorrupted version, this version propagates to the original inode (and potentially other ditto block) to restore the file. If the file cannot be recovered, the ilock operation will fail with an error code indicating the file to be corrupted.

In order to protect a high proportion of the file system, we automatically protect nodes closer to the root of the file system. Directories within 3 levels of the root are automatically protected with two ditto blocks. Directories within 6 levels of the root are automatically protected with one ditto block.

Files Changed

fs.c: Large changes to ilock, writei, and iupdate to handle updating children, writing to children, etc. New methods, ilock_trans, ilock_ext, writei_ext, cupdate, etc. were added to update children, checksums, allowing the kernel to skip checking the checksum during recovery phases, and return appropriate errors. In addition calls like iduplicate and irescue were added which perform the actual recovery.

sysfile.c: Integrated with new API from fs.c, especially utilizing returned error codes from ilock and using ilock_trans in some cases. These changes do not change the way user programs need to be written.

file.h: The inode definition was extended to include the child1, child2, and checksum fields

cat.c: The cat program was modified to accept a '-f' parameter as well. The '-f' parameter allows the user to cat a file that is corrupted. It uses the forceopen system call, which opens a file while ignoring the mismatched checksum

ls.c: ls now prints the new inode information and indicates when a file is corrupted

sh.c: You can no longer execute a corrupted file

syscall.[ch]: There were multiple system calls added (iopen, ichecksum, duplicate, forceopen) to assist in implementing and extending user programs

mkfs.c: This was modified to create ditto inodes for important user programs in the root directory (eg. ls, cat, etc). The disk was also enlarged to 2048 blocks (1 MB) and the log was enraged to (20 blocks).

Files Added

cfs.c: This external program allows us to randomly corrupt any inode on a given disk image, allowing us to test our recovery mechanism. In order to corrupt an inode we pass in the inode number and a corruption rate (ie. a corruption rate of 1000 means that we will flip each byte in the file with a probability of 1/1000). This file is compiled separately.
Example usage: ./cfs fs.img [inum] [rate of corruption]

idesignate.c: The idesignate user program allows the user to designate a file as important. A user may tell the operating system to use either 1 or 2 ditto blocks for a given file.

pchecksum.c: The pchecksum user program allows the user to print the checksum of a given file given a device and inode number. The checksum is recalculated when the user runs the program - it is not the stored checksum.

pcat.c: The pcat user program allows the user to cat a file given a device and inode number instead of a filename. This can be used to cat ditto blocks.

pinode.c: The pinode function prints the information normally shown by ls about a given file given its device and inode number

Log and Crashes

What happens if the log crashes during recovery? Or during a write to a child inode? Our changes integrate well with the existing system making recovery very simple. During writes to children, these writes are duplicated in writei within the same transaction (filewrite, which splits a write across multiple transactions, was modified to keep that into account), and thus recovery means that these changes would simply not happen altogether. The only incomplete-write type error is the same as previous cases: a write partially completes in a file, but then in this case the

file and its dittos are on the same pace.

This, however, means that the checksum will not be correct. This is very acceptable, in fact, because our system is focused on resilience and thus it will be very verbose when a file has a partially incomplete write. In a sense, it provides the user with an insight on the fact that a file has an incomplete write and does not have (all) the intended content. While ZFS does not provide the opportunity for the user to fix this corruption (error correction) in this case, it still provides the user with error detection, which is still valuable.

A user, in theory, can choose to use a function like **cat -f** to view a file anyways. User programs can be created to make a user accept a crashed file (recalculating its checksum) and continue working with it.

Now, what happens if the log crashes during the recovery of an inode? Well, an inode being recovered might be too large to fit in a single transaction, and so the recovery must take place on multiple transactions. Thus, it is possible to have a situation where an inode is partially recovered. This is not a problem, since the next time we attempt to lock an inode, we will witness a checksum mismatch and attempt to recover again. If an inode was recoverable before, and assuming that both dittos did not get corrupted in the meantime (the crash should not affect them as we do not write to dittos during recovery), we can recover it again and thus have a valid inode eventually. This is because the checksum will still mismatch, leading us to begin recovery all over again.