Project 5b: File System Integrity

Overview

In this project, you'll be changing the existing xv6 file system to add protection from data corruption. In real storage systems, silent corruption of data is a major concern, and thus many techniques are usually put in place to detect (and recover) from blocks that go bad.

Specifically, you'll do three things. First, you'll modify the code to allow the user to create a new type of file that keeps a **checksum** for every block it points to. Checksums are used by modern storage systems in order to detect silent corruption.

Second, you'll have to change the file system to handle reads and writes differently for files with checksums. Specifically, when writing out such a file, you'll have to create a checksum for every block of the file; when reading such a file, you'll have to check and make sure the block still matches the stored checksum, returning an error code (-1) if it doesn't. In this way, your file system will be able to detect corruption!

Third, for information purposes, you will also modify the `stat()` system call to dump some information about the file. Thus, you should write a little program that, given a file name, not only prints out the file's size, etc., but also some information about the file's checksums (details below).

Details

To begin, the first thing to do is to understand how the file system is laid out on disk. It is actually quite a simple layout, much like the old Unix file system we have discussed in class. As it says at the top of **fs.c** , there is a superblock, then some inodes, a single block in-use bitmap, and some data blocks. That's it!

What you then need to understand is what each inode looks like. Currently, it is quite a simple inode structure (found in **fs.h**), containing a file type (regular file or directory), a major/minor device type (so we know which disk to read from), a reference count (so we know how many directories the file is linked into), a size (in bytes), and then NDIRECT+1 block addresses. The first NDIRECT addresses are direct pointers to the first NDIRECT blocks of the file (if the file is that big). The last block is reserved for larger files that need an indirect block; thus, this last element of the array is the disk address of the indirect block. The indirect block itself, of course, may contain many more pointers to blocks for these larger files.

The change we suggest you make is very simple, conceptually. Your implementation should keep the inode structure exactly as is, but to use the slots for direct pointers as (checksum, pointer) pairs. Specifically, use each direct pointer slot (which are 4 bytes) as a 1-byte checksum and a 3-byte pointer. This limits how many disk addresses your file system can refer to (to 2^24), but that is probably OK for this project.

The simple one-byte checksum you will be computing over each block is XOR. Thus, to compute the checksum of a block, you should XOR all of the bytes of the block and store that value in the pointer to the block as described above. This is done every time a particular block is written, so that the file system keeps checksums up to date.

On subsequent reads of a block with a checksum, the file system should read in the block, compute its checksum, and compare the newly computed checksum to the stored checksum (the one stuffed into the pointer). If the stored and computed checksums match, your code should return as usual; if not, the call to `read()` should return an error (-1). Note that your file system does not make replicas of blocks, and thus does not have to recover from this problem; rather, you just signal an error and don't return corrupted data to the user.

Note also that large files will also have an indirect pointer allocated to them, and thus the indirect block will be full of direct pointers too, which also should have checksums in the manner described above.

One major obstacle with any file system is how to boot and test the system with the new file system; if you just change a bunch of stuff, and make a mistake, the system won't be able to boot, as it needs to be able to read from disk in order to function (e.g., to start the shell executable).

To overcome this challenge, your file system will support two types of files: the existing pointer-based structures, and the new checksum-based structures. The way to add this is to add a new file type (see **stat.h** for the ones that are already there like T_DIR for directories, and T_FILE for files). Let's call this type T_CHECKED (a new file type, with the numeric value of 4). Thus, for regular files (T_FILE), you just use the existing code, without checksums. However, when somebody allocates an file that has extra protection (T_CHECKED), you should use your new checksum-based code.

To create an checksum-based file, you'll have to modify the interface to file creation, adding an O_CHECKED flag (value 0x400) to the `open()` system call which normally creates files. When such a flag is present, your file system should thus create an checksum-based file, with all of the expected changes as described above. Of course, various routines deeper in the file system have to be modified in order to be passed the new flag and use it accordingly.

There is no need to do any of this for directories.

Of course the real challenge is getting into the file system code and making your checksum-based modifications. To understand how it works, you should follow the paths for the `read()` and `write()` system calls. They are not too complex, and will eventually lead you to what you are looking for. Hint: at some point, you should probably be staring pretty hard at routines like **bmap()** .

Finally, you'll have to modify the `stat()` system call to return information about the actual disk addresses in the inode (stat() currently doesn't return such information). Here is the new stat structure you should use.

This structure has room to return exactly one byte of checksum information about the file. Thus, you should XOR all of the existing checksums over blocks of the file and return that value in the `checksum` field of the stat structure.

Also create a new program, called **filestat** , which can be called like this: **filestat pathname** . When run in such a manner, the filestat program should print out all the information about a file, including its type, size, and this new checksum value. Use this to show that your checksum-based file system works.

The Code

The code (and associated README) can be found in **~cs537-1/ta/xv6/** . Everything you need to build and run and even debug the kernel is in there, as before.