Design Doc Asg4:

Ian Feekes
#1474914
ifeekes@ucsc.edu

Table of Contents:

1 Things to Note:

This project has two cases for the same type of file system:

-Block size of 512 Bytes
-Block size of 8 KibiBytes

For both of the cases, the code and implementation of the file system is quite nearly the same: only a few macros and basic allocations are changed in a few functions, but for the most part the code to write and implement the systems remains the same between the two cases - the idea is to contrast, not to write more algorithms.

To run the disk image code:

$: make
$: ./fatMain [diskName] [numBlocks]

This will initialize a FAT system that will write its data into a file with the given disk name. I initially tested this by checking the file size to make sure that BLOCKSIZE*numBLOCKS data was allocated to the file.

The diskReader file contains code for parsing a FAT disk image to conveniently test the disk images created by fatMain.c.

To test the disk image code (after creating a disk image file):

$: ./diskReader [diskName] > [outputFile]

This will redirect output into a separate file because the diskReader is designed to spill out any non-zero index in the FAT table, and with larger and larger numbers of blocks, you can get thousands of non-zero indices being printed on separate lines.

2 What I Did:

superblock.h:

Header file containing all the data fields for a super block. All are unsigned 32 bit integers corresponding with the assignment prompt

directory.h

Header file containing all the data fields for a directory. The fields consume different amounts of memory, all based on the assignment prompt

fatblock.h

Header file containing an array of pointers to data blocks. Not aptly named, it represents the File Allocation Table.

fatMain.c

Usage: ./fatMain [diskImageName] [numBlocks]

This is well-documented, and the comments within the file itself may prove equally helpful to the design doc summary.

File containing the main method for the creation of a FAT file system. Contains the function writeBlock, which is a convenient way of making the system call to find a specific block before writing to it (which is used often).

The main method parses arguments, initializes the file descriptor for the disk image, and writes dummy blocks into it to set its memory. It initializes the first block (block 0) to super and sets its fields appropriately, writing it to disk. It then makes necessary calculations for the File Allocation Table, choosing to do it by overwriting a single structure with enough indices to point to a single block, rather than simply continuing to iterate through a single larger structure with enough indices to point to each block which perhaps was a poor programming decision. It writes the FAT to disk, and then initializes a single root directory and its data fields, writing it to disk afterwards. It then frees the data structures created and closes the file, as all the necessary components of a FAT file system have been written to a disk image.

diskReader.c

Usage: /diskReader [diskImageName]

This is well-documented, and the comments within the file itself may prove equally helpful to the design doc summary. File containing a main method for testing disk creation by fatMain.c. It uses file type objects rather than file descriptors because the read system call was giving issues during the initial writing of this code.

It reads the disk image into a buffer, and uses the buffer to initialize the super block and the file allocation table, spewing them out into standard output based on the formatting. It's probably best to redirect output to a file because with larger block sizes, the file allocation table can have many pointers that can be confusing to have to parse through (but also useful).

littleBlock/bigBlock.c
Usage: sudo ./littleBlock [mountDirectory] (or) sudo ./bigBlock [mountDirectory]

Both of these files, found in the case1 and case2 directories respectively, are relatively the same in their implementation and structure: the only real difference between the two is the declaration of the macro BLOCKSIZE, which is 512B for case1 and 8192B for case2 as specified by the assignment prompt.

The biggest point of addition for these files is the loading of a pre-constructed file system that occurs before taking in the special operations in main. It sets the disk's file descriptor to open any local file called "disk" which is assumed to be a pre-constructed disk image via execution of fatMain.c, and it allocates the appropriate data fields of the superblock, the FAT table, and the initial root directory, in a manner similar to the behavior of diskReader.c's implementation.

Most of the FUSE operations are unimplemented (I commented out and deleted most of the code I was working on so that you don't have to read through meaningless code). The only operation that works to my knowledge is get_attr, which is difficult to test on its own, though it is used often.

NOTE: There are two slightly different versions of diskReader and fatMain in the case 2 directory that perform more or less the same in terms of their functionality: the main difference is the block size macro and how that plays a role in the systems that these files create.

3 What I Didn't Do:

Fuse implementation. For each case I constructed a file (littleBlock.c and bigBlock.c) that allows for the FUSE operations, however I was unable to get any of the FUSE operations to work as desired.

The most significant portions of each of the fuse files is the addition in main that allows a disk image to be loaded into memory for the virtual file system. The requirement for this to happen is that the disk image is named "disk" before littleBlock is mounted.

The implementations of FUSE operations are nothing significant past hello.c FUSE example with the exception of the main function and file_getattr.

My main excuse for this failure is that my debugging flags (both -d and -f) would crash the system (even after reinstallation), so the only logging that I could do was through the filler function.

To my knowledge, the disk image creation, reading, and loading all works with no problems.

4 How I Tested it:

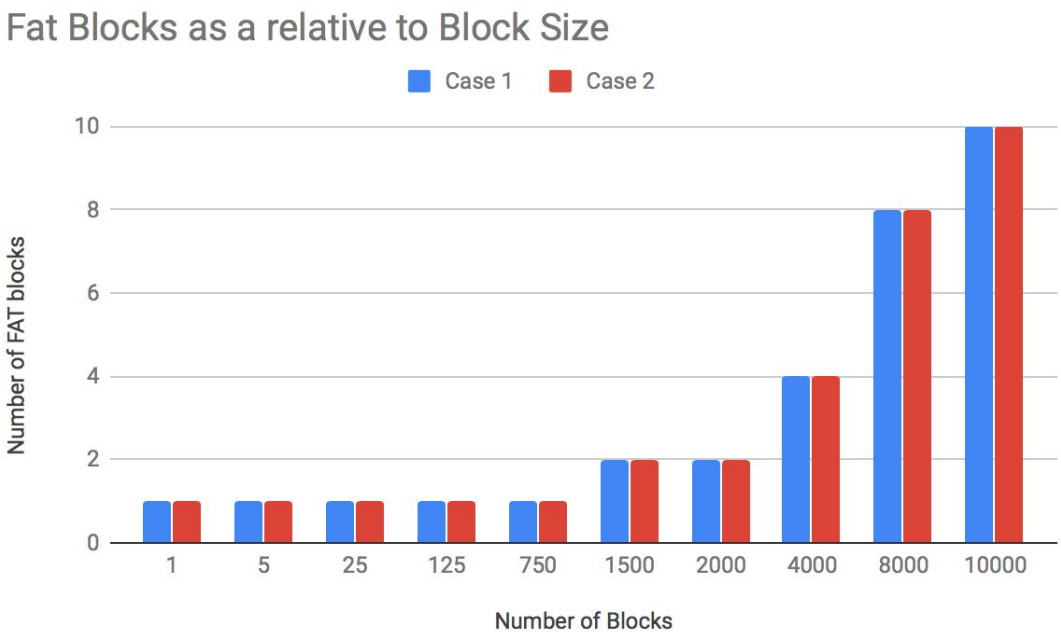
A large amount of my testing went into creating file systems for both cases using the two variations of fatMain.c, and reading the disk images created using diskReader.c to verify the data.

Since my fuse operations have very little functionality, I decided to try my best for substitution benchmarking through comparing the data of the file systems between the two cases of small and significantly larger block sizes in a file system. The results can be seen below under “What I Got.”

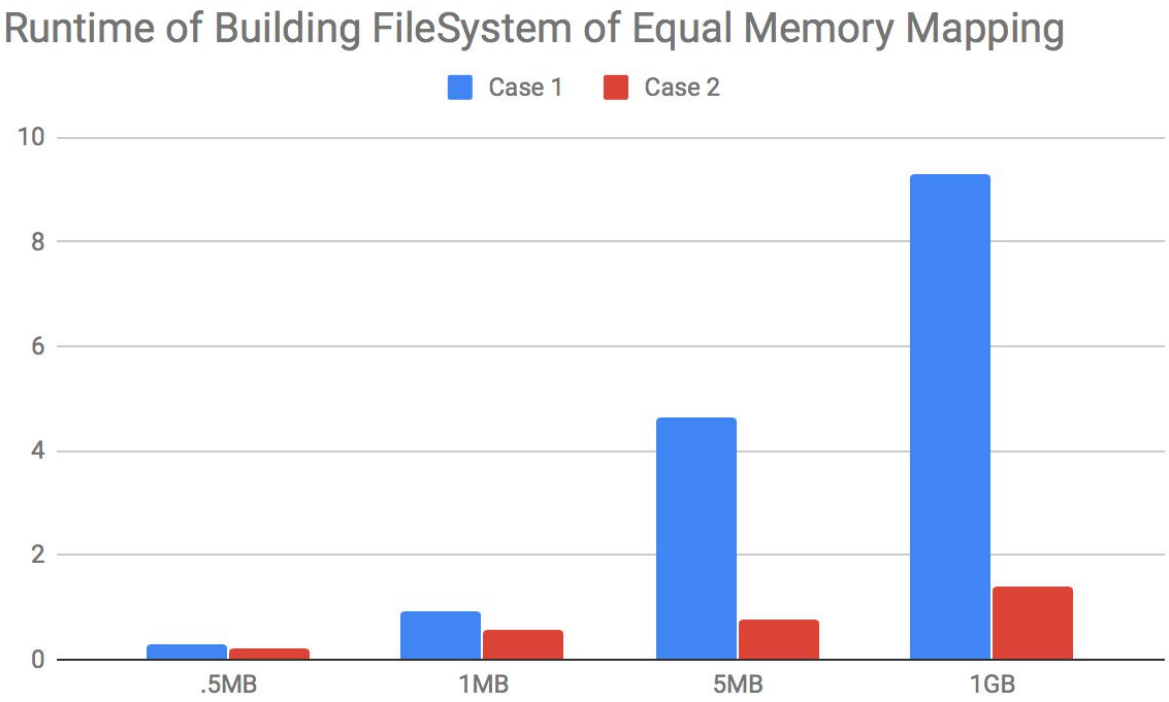
I first compared the number of blocks in usage for the FAT in both cases in an effort to compare overhead between file systems with smaller and larger block sizes. This was done simply through analyzing the output of diskReader on disk images constructed with various inputs for number of blocks. This was done for a variety of blockNumbers between the two cases between 1 and 10,000.

The second attempt at benchmarking was to compare the speeds of the two file systems being initialized to equal sizes (but not an equal number of blocks, as case2 has blocks that are 16x larger). This could be done through the command
$: time ./fatMain disk [number of blocks]
This was done for file systems of sizes .5MB, 1MB, 5MB, and 1GB.

5 What I Got:



Fat Blocks remain a constant between both cases in their relativity to block size, as is fairly obvious here. One thing to note, however, is that my ideas of extra overhead are not necessarily wrong, as the larger block size of case2 may lead to more unused memory (even with the same number of FAT blocks) because you can have the scenario of a 8KB block, with only a few bytes being used for FAT pointers, whereas the worst case scenario for case1 is far less dire.



The runtime goes with my initial assumptions: File systems with bigger block sizes tend to follow the trend of being faster, but significantly less efficient because they waste space (such as through allocating more data than needed with their large blocks), when smaller blocks get better utility but less speed. Though I didn't have any benchmarks to measure the utility side of this trend, with file systems of equal total memory, the file system with larger block size operated at a significantly faster rate, to the point where the source of error of initializing more data structures in the defense of case1 becomes less and less appealing.