CMPS111 Asgn 4

Barron Wong & John Runnberg
bwong20@ucsc.edu jrunnber@ucsc.edu

March 08, 2019

Introduction

In assignment 4 we work with FUSE to build an all-in-one filesystem.

1 Superblock

- **Blocksize** 4096B [0-4095]
    - **Magic Number** 4B [0-3]
    - **Bitmap** 4092B [4-4095]

At the start of the AOFS is the superblock. The first four bytes contains the magic number 0xFA19283E; the rest of the space in the super block is utilized for the bitmap. Since we are limiting the superblock to a single block, we can calculate the total size of the file system, by taking the product of the number of addressable blocks in the bitmap with the size of a block; this comes out to be around 134MB.

2 Blocks

- **Blocksize** 4096B [0-4095]
    - **Metadata** 376B [0-375]
        - **Struct Stat** 120B [0-119]
        - **Filename** 256B [120-375]
    - **Data** 3720B [376-4095]

The struct stat is a structure defined in the sys/stat.h file. Since this is the default structure that FreeBSD uses for file, we've decided it would make things easier to keep everything homogeneous.

Listing 1: struct stat

```
struct stat {
   __dev_t  st_dev;      /* inode's device */
   ino_t  st_ino;        /* inode's number */
   mode_t    st_mode;       /* inode protection mode */
   nlink_t   st_nlink;     /* number of hard links */
   uid_t  st_uid;       /* user ID of the file's owner */
   gid_t  st_gid;       /* group ID of the file's group */
   __dev_t  st_rdev;        /* device type */
   struct  timespec st_atim; /* time of last access */
   struct  timespec st_mtim; /* time of last data modification */
   struct  timespec st_ctim; /* time of last file status change */
   off_t  st_size;       /* file size, in bytes */
   blkcnt_t st_blocks;     /* blocks allocated for file */
   blksize_t st_blksize;   /* optimal blocksize for I/O */
   fflags_t st_flags;      /* user defined flags for file */
   __uint32_t st_gen;      /* file generation number */
   __int32_t st_lspare;
   struct timespec st_birthtim; /* time of file creation */
    .
    .
    .
```

We will be utilizing st_flag to determine if a block is the first of the file or whether it is an extension. For block pointers, we will be utlizing st_lspare, since it seems like a spare 32-bit element.

The next 256B will be reserved for the filename and the remainder will be used for data.

3 Filesystem Fucntions

FUSE allows us to modify filesystem function to implement our own filesystem in userspace. We were tasked with implementing 5 such functions:
Getattr
Create
Unlink
Read
Write

For the Getattr function we will take the filename and seek the metadata of all blocks in order until we find the file with the given name. Once found we will read and return the relevant the metadata.
To create a file we will first find the first free bit in the bitmap, contained within our superblock, and set it. Then we will seek the location of the block represented by the bit. Then we will write the relevant metadata to the front

of the block.
The Unlink function will simply unset the bits of all the blocks used by a file. In the case of a single block the file will be found by name and the location used to determine the bit to unset. If the file is longer than one block we will follow the block pointers and unset the appropriate bits along the way. We will leave whatever is written in the blocks since it can be overwritten when the space needed.
After having found the requested files first block, read will check to see if this block contains a pointer to a next block then it will transfer the data section of the block into a buffer. If the block had a next block this process repeats until otherwise.
To write to a file we will search for the first block and then follow any block pointers. Once at the end of the file we will write as much as we can to the current block and update the datasize segment of the metadata accordingly. If more data still needs to be written we will obtain a new free block, set the block pointer on the previously last block, and write metadata to the new block. We can repeat this process until all of the data is written.

4 Benchmarking

Our benchmark for this program consists of a single program called benchmark. The source file is benchmark.c and can be compiled with "make benchmark".
The benchmark will create and write to 99 files. The first 33 files will have 3000 bytes written to them, the following 33, 6000, and the final 33 will have 12000 bytes written to them–all written byte by byte. The benchmark will output the time taken to do this. After that, the benchmark will read all the files byte by byte and output the time taken to do so.
The benchmark can be used once on FreeBSD's default file system and then on our file system; the results can then be compared to see which file system preforms better.
In our test of the benchmark the FreeBSD file system was able to preform the aformentioned tasks in .124 seconds and .095 seconds, respectively. Whereas, our AFOS filesystem preformed the same tasks in .627 and .391 second, respectively

4.1 Running the Benchmark

The benchmark is made to be run in the same directory as the mnt directory and FS_FILE. The benchamrk can be compiled with "make benchmark" and run with "./benchmark". To run the benchmark on FreeBSD's file system make sure mnt is an empty directory and is not mounted. Then, outside if mnt, compile and run the benchmark. The benchmark will print the time take for writing and reading.
To run the benchmark on our file system first follow the instructions for mounting found in README. Once our file system is set up make sure FS_File is

correctly initialized and mnt is empty (not required but to keep in line with the procedure for running on FreeBSD file system). Then compile and run the benchmark. The benchmark will print the time take for writing and reading.

Conclusion

This assignment has taught us a lot about how computers and operating systems represent information. Building a file system, in concept, requires giving context and meaning to a string of bits. And in practice, it requires efficient operations that move data and maintain the abstractions.