Project #5: FATty File System

Kyu-Jin Cho

Systems Software & Architecture Lab.

Seoul National University

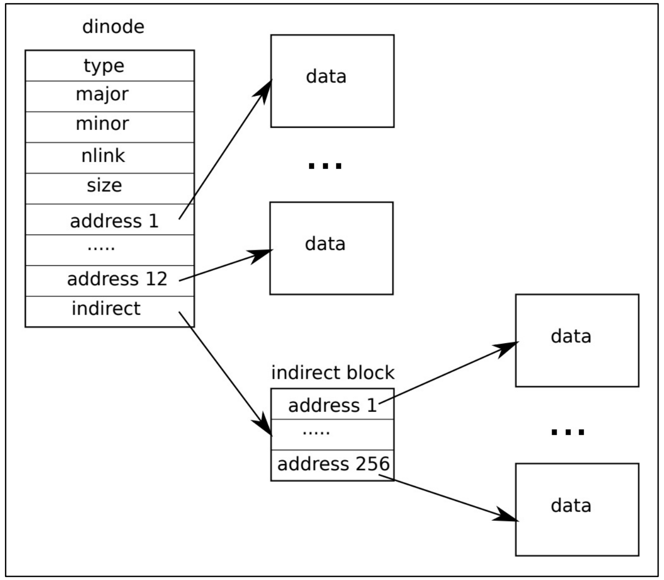
2024.06.04

The xv6 file system

- The xv6 file system provides Unix-like files, directories, and pathnames, and stores its data on a virtio disk for persistence
  - B: Boot block (1 block) -- Not used
  - S: Superblock (1 block)
  - L: Log blocks (30 blocks)
  - I: Inode blocks (13 blocks)
  - M: Free bitmap blocks (1 block)
  - D: Data blocks (1954 blocks)

```
+---+---+-------+------+---+--------------------------------------+
| B | S |   L   |  I   | M |                  D                   |
+---+---+-------+------+---+--------------------------------------+
```

< Structure of the xv6 file system>



< Representation of a file on disk>

Microsoft FAT File System

- The FAT file system, developed by Microsoft in 1977, is one of the earliest and simplest file systems
- The FAT file system exists in several versions, including FAT12, FAT16, and FAT32, each extending the maximum storage capacity and improving performance

- The FAT file system uses a table at the beginning of a disk to manage files and directories
- The table maintains pointers to the next block in a file, allowing for sequential access and easy file allocation

File Allocation Table (FAT)

- The FAT contains information on the file index, specifically the locations of the blocks belonging to each file or directory
- The FAT has an entry for each block, and each entry points to the next block number in the file
  - 0: not allocated
  - -1: EOF
  - -2: reserved block

FAT[0]	FAT[1]	FAT[2]	FAT[3]	FAT[4]	FAT[5]	FAT[6]	FAT[7]
-2	-2	3	4	7	6	9	8
FAT[8]	FAT[9]	FAT[10]	FAT[11]	FAT[12]	FAT[13]	FAT[14]	…
-1	10	-1	0	0	0	0	…

< Example of FAT>

File Allocation Table (FAT)

- The FAT contains information on the file index, specifically the locations of the blocks belonging to each file or directory
- The FAT has an entry for each block, and each entry points to the next block number in the file
  - 0: not allocated
  - -1: EOF
  - -2: reserved block

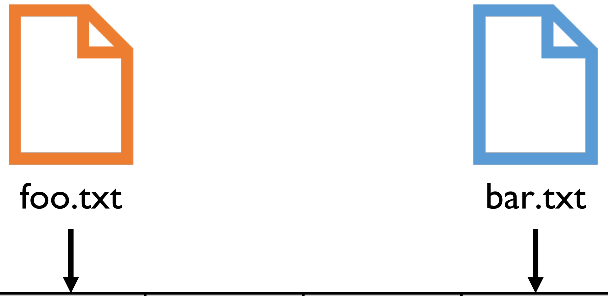
foo.txt

FAT[0]	FAT[1]	FAT[2]	FAT[3]	FAT[4]	FAT[5]	FAT[6]	FAT[7]
-2	-2	3	4	7	6	9	8
FAT[8]	FAT[9]	FAT[10]	FAT[11]	FAT[12]	FAT[13]	FAT[14]	…
-1	10	-1	0	0	0	0	…

< Example of FAT>

File Allocation Table (FAT)

- The FAT contains information on the file index, specifically the locations of the blocks belonging to each file or directory
- The FAT has an entry for each block, and each entry points to the next block number in the file
  - 0: not allocated
  - -1: EOF
  - -2: reserved block



FAT[0]	FAT[1]	FAT[2]	FAT[3]	FAT[4]	FAT[5]	FAT[6]	FAT[7]
-2	-2	3	4	7	6	9	8
FAT[8]	FAT[9]	FAT[10]	FAT[11]	FAT[12]	FAT[13]	FAT[14]	…
-1	10	-1	0	0	0	0	…

< Example of FAT>

Project#5: FATty File System

- In this project, you have to
  1. Modify the *mkfs* tool (20 points)
  2. Replace the file index structure with FAT (60 points)
  3. Implement the sync() (10 points)
  4. Design document (10 points)

- Due date is 11:59 PM, June 22 (Saturday)

FATty File System

- The FATty file system uses the file index structure that resembles that of the FAT file system
  - B: Boot block (1 block) -- Not used
  - S: Superblock (1 block)
  - L: Log blocks (30 blocks)
  - F: FAT blocks (8 blocks)
  - I: Inode blocks (4 blocks)
  - D: Data blocks (1956 blocks)

```
+---+---+-------+------+---+------------------------------------+
| B | S |   L   |  F   | I |                 D                  |
+---+---+-------+------+---+------------------------------------+
```

< Structure of the FATty file system>

FATty File System (cont'd)

- Minor changes from the FAT file system

1. The FATty file system has a magic number 0x46415459 (= "FATY")
2. We maintain only one copy of the FAT blocks for simplicity
3. Each FAT entry is encoded as a signed 32-bit integer with the following values
    - Positive values (> 0): denote the next block number
    - Zero (0): denotes the end of the file
    - Negative one (-1): indicates that the corresponding blocks are reserved (this applies to the entries for the boot block, superblock, log blocks, FAT blocks, and inode blocks)

FATty File System (cont'd)

- Minor changes from the FAT file system

4. The first block number is kept in the inode's startblk
5. The unallocated (free) data blocks are also linked together via FAT entries
   - The head of the free block list is maintained in the freehead field of the superblock
   - The total number of free blocks is stored in the superblock's freeblks field
6. During file system operations, only the in-memory versions of the superblock and FAT blocks are updated
   - To make these updates persistent, users must explicitly call the sync() system call

1. Modify the *mkfs* tool

- You should modify mkfs to set up the FATty file system
- The FAT blocks must be positioned between the Log blocks and the Inode blocks
- You should correctly initialize the free block list and the corresponding superblock fields such as freehead and freeblks

```
struct superblock {
  uint magic;        // Must be FSMAGIC
  uint size;         // Size of file system image (blocks)
  uint nblocks;      // Number of data blocks
  uint ninodes;      // Number of inodes.
  uint nlog;         // Number of log blocks
  uint logstart;     // Block number of first log block
  uint inodestart;   // Block number of first inode block
#ifdef SNU
  uint nfat;         // Number of FAT blocks
  uint fatstart;     // Block number of first FAT block
  uint freehead;     // Head of the free block list
  uint freeblks;     // Number of free data blocks
#else
  uint bmapstart;    // Block number of first free map block
#endif
};
```

2. Replace the file index structure with FAT

- Each inode only contains a pointer (startblk) to the first data block
- The subsequent block locations should be looked up in the FAT
- When a data block is allocated or deallocated, ensure that the superblock's freeblks value is updated accordingly
  - The skeleton code includes functionality to print this value whenever you press ^f (ctrl-f) in the console
  - This value will be checked to determine whether your implementation has space leaks or not during various file system operations

```
struct dinode {
  short type;           // File type
  short major;          // Major device number (T_DEVICE only)
  short minor;          // Minor device number (T_DEVICE only)
  short nlink;          // Number of links to inode in file system
  uint size;            // Size of file (bytes)
#ifdef SNU
  uint  startblk;       // First block number
#else
  uint addrs[NDIRECT+1];   // Data block addresses
#endif
};
```

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ Free blocks: 1266
```

3. Implement the sync()

- Your task is to implement a new system call named sync()
  - The system call number of sync() is already assigned to 22

- Return value
  - 0 (always success)

- The role of the sync() system call is to write the contents of the superblock and FAT blocks to the disk to make them persistent
- You don't need to care about sudden power failures during the sync() system call

4. Design Document

- You need to prepare and submit the design document for your implementation
- You should explain what you have considered, and what you have done

- Requirements
  - New data structures
  - Algorithm design
  - Testing and validation

Restrictions

- Your implementation should pass usertests on multi-processor RISC-V systems (i.e., CPUS > 1)
  - You need a synchronization for accessing superblock and FAT blocks
- There should be no space leaks in the file system
- You only need to modify those files in the ./kernel and ./mkfs directory
  - Changes to other source code will be ignored during grading
- Please remove all the debugging outputs before you submit

Tips

- Read Chap. 8 of the xv6 book to understand the file system in xv6
- For your reference, the following roughly shows the amount of changes you need to make for this project assignment
- Each "+" symbol indicates 1~10 lines of code that should be added, deleted, or altered

```
kernel/defs.h     | +
kernel/fs.c       | +++++++++++++++++
mkfs/mkfs.c       | ++++++++++++++++++
```

Skeleton Code

- Skeleton Code
  - You should work on the pa5 branch of the xv6-riscv-snu repository as follows:

```
$ git clone https://github.com/snu-csl/xv6-riscv-snu
$ git checkout pa5
```

  - The pa5 branch includes a sample FATty file system image, fs-fatty.img. Using this image file, you can start Part 2 without completing Part 1 of this project. If you want to use this image file, copy it to fs.img before running xv6
  - The current skeleton code is unable to build the kernel image due to the changes in the inode and superblock structures

Notification

- Due
  - 11:59 PM, June 22 (Saturday)

- Submission
  - Run the make submit command to generate a tarball named xv6-pa5-{STUDENTID}.tar.gz in the xv6-riscv-snu directory
  - Upload the compressed file to the submission server
  - The total number of submissions for this project will be limited to 50
  - Only the version marked FINAL will be considered for the project score

Thank you!