CS 310 FAT Filesystem Driver

Spring 2021

1 Getting Started

In this homework, you're going to be writing a FAT filesystem driver for your operating system. The end goal is to be able to read an arbitrary file from the SD card into memory. I will provide you with an SD card driver that can read sectors from the disk into memory. The job of the FAT FS driver is to parse the on-disk datastructures (superblock and inodes) to find the data blocks corresponding to a given filename and load that file into a buffer.

Deliverables: You are going to write three functions to access a FAT filesystem on your SD card. You should call all three of these functions from `kernel_main` to test them—open a file and read its contents into a memory buffer.

1. `fatInit`: Initializes the FAT filesystem driver by reading the superblock (aka boot sector) and FAT into memory.
2. `fatOpen`: Opens a file in a FAT filesystem on disk.
3. `fatRead`: Reads data from a file into a buffer

2 Getting Started: Reading FAT Data Structures from Disk

Below I have pasted a data structure that lists all the fields in the boot sector and the BIOS information block. Your job is to read one disk sector into memory (by calling the SD driver) and parse the fields. You can read blocks from the disk drive by calling the SD driver included in `sd.c`:

3 Opening a File

Suppose we are trying to find the location of the file `/BIN/BASH`[1] on our FAT filesystem. Opening a file basically consists of searching for the file's root directory entry (RDE) in the root directory region. Every file and every directory that live in the root of the FAT FS are pointed to by an RDE. In the diagram below, the `BIN` directory is a child of `/` (the root of the filesystem). `BIN`, `BOOT`, and all other children of the root are assigned RDEs in the root directory region of the filesystem. Each RDE contains

- `name`
- `attributes` (read-only, hidden, file type, etc.)
- `cluster`: the location of the beginning of the file
- `size`

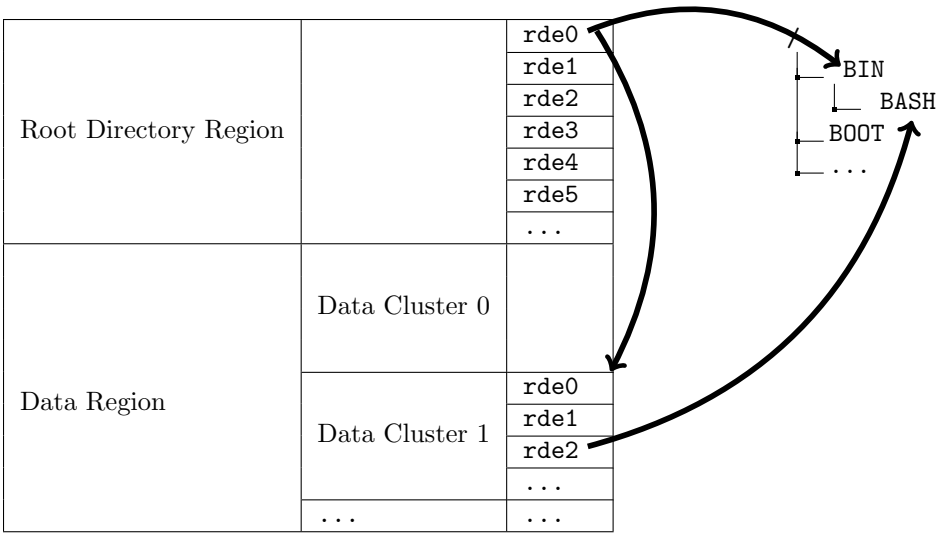
In the case of directory `BASH`, its RDE `cluster` points to data block 1 in the filesystem. We will know that it is a directory (and not a file) by parsing its `attributes` field in the RDE[2]. Data cluster 1 contains a bunch of RDEs, each of which points to a file or subdirectory in `/BIN`. To find the location on disk of `BASH`, we must loop through each RDE in data cluster 1, searching for one that has the name `BASH`. In the diagram below, `rde2` links to the `BASH` file. The `cluster` field of `rde2` will point to the data cluster that contains the data for the `BASH` file.

[1] All file and directory names are stored in capital letters in the FAT FS. You can use the C library function `toupper` to convert a filename to uppercase before searching for it on disk.

[2] Details of the RDE structure are available on the Wikipedia article about the FAT filesystem:
`https://en.wikipedia.org/wiki/Design_of_the_FAT_file_system#Directory_entry`

```
#include "sd.h" // See this file for function prototype of ata_lba_read()

struct boot_sector *bs;
char bootSector[512]; // Allocate a global array to store boot sector
char fat_table[8*SECTOR_SIZE];
unsigned int root_sector;

int fatInit() {
  sd_readblock(0, bootSector, 1); // Read sector 0 from disk drive into bootSector array
  bs = bootSector; // Point boot_sector struct to the boot sector so we can read fields

  // Print out some of the elements of the BIOS information block using rprintf...

  // TODO: Validate the boot signature = 0xaa55
  // TODO: Validate fs_type = "FAT12" using strcmp
  // TODO: Read FAT table from the SD card into array fat_table
  // TODO: Compute root_sector as:
  //       num_fat_tables + num_sectors_per_fat + num_reserved_sectors + num_hidden_sectors
}
```

```
// Boot sector data struct. Put this either above main() or in its own header file.

struct boot_sector {
    char code[3];
    char oem_name[8];
    uint16_t bytes_per_sector;
    uint8_t num_sectors_per_cluster;
    uint16_t num_reserved_sectors;
    uint8_t num_fat_tables;
    uint16_t num_root_dir_entries;
    uint16_t total_sectors;
    uint8_t media_descriptor;
    uint16_t num_sectors_per_fat;
    uint16_t num_sectors_per_track;
    uint16_t num_heads;
    uint32_t num_hidden_sectors;
    uint32_t total_sectors_in_fs;
    uint8_t logical_drive_num;
    uint8_t reserved;
    uint8_t extended_signature;
    uint32_t serial_number;
    char volume_label[11];
    char fs_type[8];
    char boot_code[448];
    uint16_t boot_signature;
}__attribute__((packed));
```



4 Creating a Disk Image with a FAT FS

To test your FAT FS driver, you'll need to create a disk image with a filesystem and put some files in it that your driver can read. The PiOS Makefile partly does this for you—it creates `rootfs.img`, writes a FAT 12 FS to it and creates some directories.

4.1 Disk Images

When you boot your PiOS inside qemu, you need to give it a virtual disk to use instead of an SD card that is on the real Raspberry Pi. That virtual disk is called a disk image. It is a binary file that has exactly the same contents as the SD card would have. For our purposes, it only needs to be 16 or 32 mbytes. To create an empty file filled with zeros, we can use the following command:

```
dd if=/dev/zero of=rootfs.img bs=1M count=16
```

This command uses the program disk dump (`dd`) to copy bytes from one file to another file. The input file (specified by `if=`) is `/dev/zero`, which is a special (fake) file that gives the value zero every time you read it. The output file (specified by `of=`) is `rootfs.img`, the new disk image we will create. The `bs=1M` argument means we will copy data from the input file in blocks of 1 megabyte. The `count=16` argument means we will copy a total of 16 blocks, resulting in a 16 megabyte file filled with zeros. We can treat this newly-created disk image the same way as any physical disk on the system (for example, `/dev/sda`). We now need to write a FAT filesystem to our empty disk image using the `mkfs.fat` tool:

```
mkfs.fat -F16 rootfs.img
```

This command sets up the data structures for a FAT 16 filesystem on our disk image, including the boot sector, the root directory entries, and the file allocation tables. Next, we will mount the filesystem image onto our Linux system:

```
sudo mount rootfs.img /mnt/disk
```

As we discussed in class, the mount command causes the contents of `rootfs.img` to be connected as a subtree to our main Linux filesystem. We need to use `sudo` to mount a disk image because mounting is a privileged operation. If we create a new file inside `/mnt/disk/`, the contents of that file will actually be written to `rootfs.img`. Let's create a test file:

```
sudo echo "This is a test file" >> /mnt/disk/testfile.txt
```

Now we can unmount rootfs.img, which disconnects it from the main Linux filesystem:

```
sudo umount /mnt/disk
```

Use `hexdump` to inspect the raw contents of `rootfs.img`:

```
hexdump -C rootfs.img | less
```