CIS 5480 Recitation 4 - File System & pennfat

Welcome to recitation!!! 🦛

Name: _____ **PennID: _____**

1. Designing open

You are now tasked with designing open for an inode-based filesystem. From the user's perspective, calling open returns a file descriptor that can later be used with read and write. From the filesystem's perspective, open must resolve a path name, locate an inode, and initialize per-open state.

Assume:

- Directories store mappings from file names to inode numbers
- The filesystem maintains a global open file table (OFT)
- Each process maintains a per-process file descriptor table

Questions:

- Outline the sequence of steps required to implement 'open(path, flags)', including path traversal, inode lookup, permission checks, allocation of open-file state. This is the analogue of s_open(), which you will implement in PennOS (and calls k_open(), which you will implement in PennFAT).

- Where should the current file offset be stored: the inode, the directory entry, the open file table entry, or the file descriptor table? Justify your answer by considering what should happen when the same file is opened multiple times.

- Suppose two processes simultaneously call write(path, ...). Identify one race condition that can occur and describe one mechanism the filesystem could use to prevent it. (We haven't learned about these yet so just speculate what you should do.)

2. Hexdumps are Scary(?)

Let's practice reverse engineering a file system directly from its raw bytes so we don't get so scared anymore.

In this question, you will be shown two PennFAT hexdumps, one above the other. Each hexdump represents the beginning of a different PennFAT file system image.Remember, each PennFAT entry is 16 bytes.

Your task is to extract structural information purely from the hex.

2.1 Before We Begin: Endianness

Sometimes, we look at integers displayed in pure hex and are confused by their representation in the terminal. Let's go ahead and disambiguate this.

Questions:

- What is the difference between big endian and little endian?

On most machines, we store multi-byte values in little endian. And the hex dump will print values from the lowest address first, from left to right.

If a 2-byte value appears in the start of the hexdump as:

```
0x00 0x10
```

What does this tell us about the block sizes and the number of FAT blocks?

2.2 PennFAT Hexdumps

The following are two PennFAT file system images shown as hexdumps. They represent two different file system configurations.

PennFAT Image A

```
00000000 04 20 ff ff 00 00 00 00 00 00 00 00 00 00 00 00 |. ..............|
00000010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
*
1001e000
```

PennFAT Image B

```
00000000 00 01 ff ff 00 00 00 00 00 00 00 00 00 00 00 00 |................|
00000010 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
*
00008000
```

Throughout the questions below, clearly indicate whether you are referring to Image A or Image B.

2.3 Reverse Engineering the FAT Region

Questions:

How many blocks does the FAT occupy for both hexdumps?

What is the size (in bytes) of each block?

What **mkfs-style command** would you run in the PennFAT assignment to create each of these two file systems?

2.4 The FAT Finale

Now, take a look at the following hex dump of the FAT after some modifications. There is only one block in the FAT but you should have known that by now.

```
00000000 00 01 ff ff ff ff 04 00 05 00 06 00 07 00 08 00
00000010 09 00 0a 00 0b 00 0c 00 0d 00 0e 00 0f 00 10 00
00000020 11 00 12 00 13 00 14 00 15 00 16 00 17 00 18 00
00000030 19 00 1a 00 1b 00 1c 00 1d 00 1e 00 1f 00 20 00
00000040 21 00 22 00 23 00 24 00 25 00 26 00 27 00 28 00
00000050 29 00 2a 00 2b 00 2c 00 2d 00 2e 00 2f 00 30 00
00000060 31 00 32 00 33 00 34 00 35 00 36 00 37 00 38 00
00000070 39 00 3a 00 3b 00 3c 00 3d 00 3e 00 3f 00 40 00
00000080 41 00 42 00 43 00 44 00 45 00 46 00 47 00 48 00
00000090 49 00 4a 00 4b 00 4c 00 4d 00 4e 00 4f 00 50 00
000000a0 51 00 52 00 53 00 54 00 55 00 56 00 57 00 58 00
000000b0 59 00 5a 00 5b 00 5c 00 5d 00 5e 00 5f 00 60 00
000000c0 61 00 62 00 63 00 64 00 65 00 66 00 67 00 68 00
000000d0 69 00 6a 00 6b 00 6c 00 6d 00 6e 00 6f 00 70 00
000000e0 71 00 72 00 73 00 74 00 75 00 76 00 77 00 78 00
000000f0 79 00 7a 00 7b 00 7c 00 7d 00 7e 00 7f 00 ff ff
```

Based on this, how many files are on the file system, how large is the largest file? You will need to implement something just identical to this, so it is imperative that you understand how to read these hexdumps.