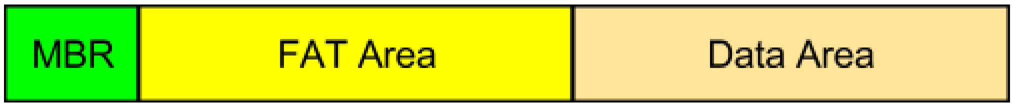
CS 3620
Operating Systems
Due: Dec 9th

Homework 4

As part of this homework, you will implement a file system that resembles FAT. The file system will be stored in a binary file within the current directory. Overall, the file system has the following setup:



The file is organized in clusters that are composed of multiple sectors. The number of sectors per cluster and the size of sectors (in bytes) are set during the formatting process.

The MBR is guaranteed to start on the first cluster of the "disk". The MBR contains the following information:

- *sector_size* (2 bytes) - the size of a sector in bytes (must be at least 64 bytes)
- *cluster_size* (2 bytes) - the size of the cluster in sectors (must be at least 1)
- *disk_size* (2 bytes) - the size of the disk in clusters
- *fat_start*, *fat_length* (2 bytes) - the start/length of the FAT area on the disk
- *data_start*, *data_length* (2 bytes) - the start/length of the Data area on the disk
- *disk_name* - the name of the disk. The name is guaranteed not to Note that this file system supports a single partition on a single disk.

The FAT area is used to keep track the allocation of clusters on disk. The FAT will contain one entry for each cluster in the data area. The value of the FAT entry is a 16-bit integer that either indicates that the cluster is unallocated (0xFFFF) or points to the next cluster that is allocated to a file/directory. The pointer is calculated relative to the start of the data area. Thus, a value of 0 indicates the first cluster in the data area that will be physically stored at *data_start*. In general, a value of *x* indicates cluster *x* that will be stored physically in cluster *data_start* + *x*.
The root directory is guaranteed to occupy the first cluster in the data area. The directory and files are stored in entries that have the following format:

- *entry_type* (1 byte) - indicates if this is a file/directory (0 - file, 1 - directory)
- *creation_time* (2 bytes*)* - format described below
- *creation_date* (2 bytes*)* - format described below
- *length of entry name* (1 byte)
- *entry name* (16 bytes) - the file/directory name
- *size* (4 bytes) - the size of the file in bytes. Should be zero for directories:
- *pointers to children files or directories*
    - *pointer type* (1 byte) - (0 = pointer to a file, 1 = pointer to a directory, 2 = pointer to another entry describing more children for this directory)

    - reserved (1 byte)
    - *start_pointer* (2 bytes) - points to the start of the entry describing the child

The fixed sized fields of the data structure (i.e., all fields excluding the pointers to the children files and directories) forms a entry_t type that is later used in fs_ls function.
The number of pointers to children files or directory depends on the size of the sector and cluster. This number is computed such that the above data structure fills an **entire cluster.**

The date and time are formatted as follows.

The date stamp is a 16-bit field that is basically a date relative to the epoch of 01/01/1980. Here is the format (bit 0 is the LSB of the 16-bit word, bit 15 is the MSB of the 16-bit word):

- Bits 0–4: Day of month, valid value range 1-31 inclusive.
- Bits 5–8: Month of year, 1 = January, valid value range 1–12 inclusive.
- Bits 9–15: Count of years from 1980, valid value range 0– = 127 inclusive (1980–2107).

Time Format: A time stamp is a 16-bit field that has a granularity of 2 seconds. Here is the format (bit0 is the LSB of the 16-bit word, bit 15 is the MSB of the 16-bit word).

- Bits 0–4: 2-second count, valid value range 0–29 inclusive (0 – 58 seconds).
- Bits 5–10: Minutes, valid value range 0–59 inclusive.
- Bits 11–15: Hours, valid value range 0–23 inclusive.

The valid time range is from Midnight 00:00:00 to 23:59:58. The words storing the date and time are in big-endian format.

As part of the assignment you will have to implement the following functions:

Format: The format function will have the following signature.

```
void format(uint16_t sector_size, uint16_t cluster_size, uint16_t disk_size)
```

The function will initialize the MBR and FAT data structures. The root directory should also be created. The sector size is in bytes, cluster size in sectors, and disk size in clusters.

Load disk: The function will load the MBR, FAT, and data from the disk file. This mechanism allows you to continue to work on a disk that you previously formatted and added contents to it.

```
void load_disk(char *disk_file)
```

The disk_file includes the file name of the disk.

Directory operations: You will have to support the following directory operations.
Opendir returns an integer that returns a handler to the directory pointed by absolute path.

```
int fs_opendir(char *absolute_path)
```

You can create a directory using the mkdir function. The dh argument is the directory handler obtained using opendir. The name of the subdirectory is provided as the second argument (child_name).

```
void fs_mkdir(int dh, char *child_name)
```

You can inspect the children of a directory using the `ls` function. Similarly, the `dh` argument is the directory descriptor. The `ls` function will return the pointer to the next child. When no more children exist, it will return null.

```
      entry_t *fs_ls(int dh, int child_num)
```

The entry_t data structure refers fixed sized fields that describe files and directories. This includes the following fields: entry_type, creation_time, creation_date, length of entry name, entry name, and size. You will return the entry_t associated with the child indicated by child_num. If the directory has fewer children than child_num, the function will return null.

Extra credit (20 points):
File operations: The following operations should be supported.
Open/close a file the file at the absolute path for reading (mode=”r”) or writing (“w”). A file is created by opening it in “w” mode. Opening a file in “r” mode would result in an error.

```
      int fs_open(char *absolute_path, char *mode)
      int fs_close(int fh)
```

Write/Read a buffer to/from a file. The stream is the file descriptor obtained using fs_open. The count is the number of bytes to read/written.

```
int fs_write( const void *buffer, int count, int stream );
int fs_read( const void *buffer, int count, int stream );
```

Your implementations of the above functions should make use of the linux open, close, read, seek, and write functions.

Extra credit (5 points):
For extra credit, define and implement functions to remove directories and files.