FAT Filesystem Demo

USAGE: `./demo_runner <filename> <data to append>`

The script `demo_runner` demonstrates a basic implementation of a FAT16 filesystem, which will be used on Domecracker and Liquid Courage III to write telemetry to an SD card during the flight.

Any string provided as the second argument to `demo_runner` will be appended to the file with the given `filename`. If `filename` does not exist, it will be created. After this is done, text will display the current status of every file within the filesystem.

Arguments:

- `<filename>`: the filename of the file to which to append data. The filename must be a maximum of 8 characters, followed by an optional three character file extention, seperated by a dot. the behavior of this script outside of these parameters is undefined, and might break everything.
- `<data to append>`: the data which to append to the given file. This should be a single string surrounded by quotation marks

API Description

The FAT filesystem implementation provides a simple API with three functions:

- `BPB* init_filesystem()`: Returns a pointer to a BPB (Bios Parameter Block) struct with metadata about the filesystem, this structure is passed into the other functions to perform operations on the filesystem.
- `FileHandle* open_file(char* filename, BPB *bpb, bool cf)` Returns a pointer to a FileHandle (a structure containing metadata about the file), it requires a filename, a BPB struct, and a boolean indicating whether to create the file if not found.
- `bool write_file(FileHandle *file, uint8_t* data, uint32_t n_bytes, BPB* bpb)` Writes to a file given a FileHandle, a buffer with data, the number of bytes to write, and the BPB.

This API is designed to be minimal, yet serve all of our needs for the telemetry of our rocket.

Implementation Details

- `init_filesystem` is the simplest of the implemented functions. All it does is read from the Bios Parameter Block of the filesystem.
- `open_file` is dependent on two helper functions: `find_file` and `create_file`. `find_file` loops through the root directory entries until it finds the entry with the provided filename or sees that no more entries exist. `create_file` relies on `find_free_cluster` to find the first free cluster available, then it allocates a directory entry pointing to said cluster, and sets the FAT entry corresponding to the cluster to `0xffff`, meaking it is the allocated last cluster in the chain.
- `write file` is the most complex of the three functions. While there is still data to write, data is written one sector at a time. At the end of each sector write, it is checked whether we've reached the end of the current cluster, and if so, we go to the next allocated cluster or allocate one if no next cluster exists. The file size of the root directory entry for the file is also updated.