

The Assignment

We have to implement a file system from scratch, given very few structural requirements. This leaves the assignment nicely open for us. I have furthermore taken the liberty of using some C++ in this assignment, mainly for memory allocation, string parsing, and the freedom to declare variables as they are needed (as opposed to having all at the start of a function). This makes the code more readable for others, and faster for me to write.

The Implementation

I have not included `disc.c` and `disc.h`, since they are virtually unmodified. There are a few more includes in `disc.h`, but all from the standard libraries. Important difference is the use of C++:

I keep track of free space using a bitset from C++. This provides quick and efficient access to search for free space, while keeping track of the total amount of used sectors. The only quirk is when loading and saving the bitset, as seen in `misc.cpp`'s functions `AllocateFMap()` and `UpdateFMap()`.

The path is parsed by `misc.cpp`'s `explode()` function, that returns a `vector<string>` of path components. This makes it very easy to `for()` loop through the components to verify the path, instead of having to use recursion.

`filesystem.h` (listing 3)

The basic overall header file, where the two all-important structs are defined. `FSEntry` is a single entry in the FAT, and can be either a file or a directory, stored in `flags`. Technically it can be both, but it has become conventional to keep directories as meta-data only. Furthermore, an entry can be marked as deleted for later undeletion (provided nobody has overwritten its data). The structure is very simple, and a directory is not in any way special. The fields `start`, `stop` are direct sector indices, while `length` is the size in byte.

```
typedef struct _FSEntry {
    unsigned char flags;
    unsigned int start, stop;
    unsigned int length;
    unsigned int parent;
    char name[16];
} FSEntry;

typedef struct _File {
    unsigned int file;
    unsigned int pos;
} File;
```

Code: important structs

No directory listing is saved, meaning that the FS must rely on the `parent` field. This is slow as it has to loop through all FAT entries just to find a certain file. It could very quickly be improved with a few arrays or multimaps. The only advantage of saving just the parent is that a directory can have unlimited direct children, and for this small file system it is no big deal to loop through the entries.

The `File` structure is also as basic as can be. It keeps track of which file that is opened, and which position this descriptor will access next. The book mentioned keeping a global and a per-process table of opened files, but seeing as how this is a single process file system I didn't implement that.

filesystem.cpp (listing 2)

The functions in this file are pretty much as prescribes. Fixed a few `int` to `unsigned int`, and added `dm14_mkfile()` to help create files for `dm14_open()`, much like Linux's `touch` command works.

When a non-existing file is opened, and thus created, it will have a length of 0. No space will be allocated for the file at this time. This means that a 0-length file only takes up a single FAT entry, and nothing more. Upon the first write, space will be allocated to the file. Initially enough to cover the requested write length, plus 2 extra sectors for growth. To accommodate this way of doing things, the file descriptor's internal position can be set to any value at any time. Only actual reading and writing will verify the position.

However, I have not implemented dynamic growth (meaning: once allocated, a file cannot extend beyond its allocation). Nor have I even considered implementing file-fragmentation support.

`dm14_delete()` handles recursive deletion of subfolders and files without any problems.

misc.cpp (listing 1)

I made some loading and saving functions for the fun of it, to see if I could create a file tree, save it, load it, and work from where it left off. This works quite well, since `AllocateFAT()` and `AllocateFMap()` will read any existing data. The map of free space is saved to disk also, but is really not used from the disk since it is re-generated from the FAT on load. It is good to compare the re-generated map against the saved map to catch FAT corruption, though.

`AtomicUpdateFAT()` provides a way to atomically save a single changed FAT entry, and will automatically determine if the FAT entry is stored across multiple sectors. If not, only one sector will be written. It does assume a FAT entry is smaller than a sector, though, but that should not be a problem in any case. The FAT is updated in this way after every successful `dm14_write()`, `dm14_mkdir()`, and `dm14_mkfile()`.

`SearchFMap()` will search for a spot of sequentially available sectors.

The Testing Phase

The testing output functionality is scattered across the functions and files, but the real testing takes place in `main()`.

As a pre-test, 3 folders are created as `/folder/subfolder/subsub/`.

#1: The first test is to see if `dm14_write()` can correctly handle writing across multiple sectors, and also resume writing from where it left off. This works correctly.

```
Disk has 28 FAT entries, of 36
byte each.
```

```
Will write 1608 byte to fd 0
Writing to entry 4 at pos 67
Filename file with parent 2
Start sector 3, stop sector 9
Wrote 445 byte to sector 3
Wrote 512 byte to sector 4
Wrote 512 byte to sector 5
Wrote 139 byte to sector 6
```

```
Will write 804 byte to fd 0
Writing to entry 4 at pos 1675
Filename file with parent 2
Start sector 3, stop sector 9
Wrote 373 byte to sector 6
Wrote 431 byte to sector 7
```

Test 1

#2: This checks if `dm14_read()` can correctly read, and also read across multiple sectors. Amazingly enough, it can.

#3: Runs a loop that will continue to write until the file is full. This works as expected, with error code `ERR_EOF` (-11, End Of File).

```
Will write 12 byte to fd 0
Writing to entry 5 at pos 507
Filename file with parent 1
Start sector 10, stop sector
14
Wrote 5 byte to sector 10
Wrote 7 byte to sector 11
```

#4: This test will write a file as large as possible, trying to fill the disk. It returns an `ERR_EOF` as before, though, since the allocation is smart enough to know you can never allocate beyond disk size. So it works as expected.

```
Will write 12 byte to fd 0
Writing to entry 5 at pos 0
Filename file with parent 1
Start sector 10, stop sector
14
Wrote 12 byte to sector 10
```

#5: A simple test of the directory listing functionality. It outputs the columns `parent,name,type,size`. Then it executes a `delete` for `/folder/`, and that recurses nicely through the subfolders and files. Works as expected.

```
Will read 12 byte from fd 0
Reading from entry 5 at pos 0
Filename file with parent 1
Start sector 10, stop sector
14
Read 12 byte
Read 'Hello World'
```

```
Will read 8 byte from fd 0
Reading from entry 5 at pos
511
Filename file with parent 1
Start sector 10, stop sector
14
Read 1 byte
Read 7 byte
Read 'o World'
```

Test 2

#6: Tests whether the various open and mkdir functions will actually detect when there isn't more available entries. Then it wipes everything again. Also shows the recursive deletion once more. The various functions (`dm14_mkdir()` and `dm14_open()`) correctly report errors when their respective limits have been reached. The test also performs a `dm14_format()` in between the saturation tests, but this function cannot fail. Overall, it works as expected.

And that is actually that. The test look simple enough in themselves, but they do get around to properly verifying that the file system works as it should. Partly because not much can really go wrong in the first place, but also because the functions are highly inter-dependent. If pretty much any of these functions did not perform as expected, the whole file system would crash and burn instantly. So performing these relatively basic open/read/write/delete tests have indirectly tested the complex functionality behind it all.

```
Will write 402 byte to fd 0
Writing to entry 5 at pos 0
Filename file with parent 1
Start sector 10, stop sector 14
Wrote 402 byte to sector 10

Will write 402 byte to fd 0
Writing to entry 5 at pos 402
Filename file with parent 1
Start sector 10, stop sector 14
Wrote 110 byte to sector 10
Wrote 292 byte to sector 11

Will write 402 byte to fd 0
Writing to entry 5 at pos 804
Filename file with parent 1
Start sector 10, stop sector 14
Wrote 220 byte to sector 11
Wrote 182 byte to sector 12

Will write 402 byte to fd 0
Writing to entry 5 at pos 1206
Filename file with parent 1
Start sector 10, stop sector 14
Wrote 330 byte to sector 12
Wrote 72 byte to sector 13

Will write 402 byte to fd 0
Writing to entry 5 at pos 1608
Filename file with parent 1
Start sector 10, stop sector 14
Wrote 402 byte to sector 13

Will write 402 byte to fd 0
Writing to entry 5 at pos 2010
Filename file with parent 1
Start sector 10, stop sector 14
Wrote 38 byte to sector 13
Wrote 364 byte to sector 14

Will write 402 byte to fd 0
Writing to entry 5 at pos 2412
Filename file with parent 1
Start sector 10, stop sector 14
Wrote 148 byte to sector 14
Error: -11
```

Test 3

```
Will write 1040384 byte to fd 0
Writing to entry 6 at pos 0
Filename bigfile with parent 0
Could not allocate 2035 sequential
sectors. Total available: 2033
Could not allocate 2034 sequential
sectors. Total available: 2033
Start sector 15, stop sector 2047
Wrote 512 byte to sector 15
Wrote 512 byte to sector ...
Wrote 512 byte to sector 2046

Will write 1040384 byte to fd 0
Writing to entry 6 at pos 1040384
Filename bigfile with parent 0
Start sector 15, stop sector 2047
Wrote 512 byte to sector 2047
Error: -11
```

Test 4

```
Listing of /
0 folder Dir 0
0 bigfile File 1040896

Listing of /folder/
1 subfolder Dir 0
1 file File 3091

Listing of /folder/subfolder/
2 subsub Dir 0
2 file File 2479

Deleted Dir: /folder/subfolder/subsub
Deleted File: /folder/subfolder/file
Deleted Dir: /folder/subfolder
Deleted File: /folder/file
Deleted Dir: /folder
```

Test 5

```

Saturated descriptors after 17 Listing of /folder/ Deleted Dir: /folder/dir13
open. Error -8 1 dir10 Dir 0 Deleted Dir: /folder/dir14
1 dir11 Dir 0 Deleted Dir: /folder/dir15
Listing of / 1 dir12 Dir 0 Deleted Dir: /folder/dir16
0 dir10 Dir 0 1 dir13 Dir 0 Deleted Dir: /folder/dir17
0 dir11 Dir 0 1 dir14 Dir 0 Deleted Dir: /folder/dir18
0 dir12 Dir 0 1 dir15 Dir 0 Deleted Dir: /folder/dir19
0 dir13 Dir 0 1 dir16 Dir 0 Deleted Dir: /folder/dir20
0 dir14 Dir 0 1 dir17 Dir 0 Deleted Dir: /folder/dir21
0 bigfile File 1040896 1 dir18 Dir 0 Deleted Dir: /folder/dir22
0 dir15 Dir 0 1 dir19 Dir 0 Deleted Dir: /folder/dir23
0 dir16 Dir 0 1 dir20 Dir 0 Deleted Dir: /folder/dir24
0 dir17 Dir 0 1 dir21 Dir 0 Deleted Dir: /folder/dir25
0 dir18 Dir 0 1 dir22 Dir 0 Deleted Dir: /folder/dir26
0 dir19 Dir 0 1 dir23 Dir 0 Deleted Dir: /folder/dir27
0 dir20 Dir 0 1 dir24 Dir 0 Deleted Dir: /folder/dir28
0 dir21 Dir 0 1 dir25 Dir 0 Deleted Dir: /folder/dir29
0 dir22 Dir 0 1 dir26 Dir 0 Deleted Dir: /folder/dir30
0 dir23 Dir 0 1 dir27 Dir 0 Deleted Dir: /folder/dir31
0 dir24 Dir 0 1 dir28 Dir 0 Deleted Dir: /folder/dir32
0 dir25 Dir 0 1 dir29 Dir 0 Deleted Dir: /folder/dir33
0 dir26 Dir 0 1 dir30 Dir 0 Deleted Dir: /folder/dir34
0 dir27 Dir 0 1 dir31 Dir 0 Deleted Dir: /folder/dir35
0 dir28 Dir 0 1 dir32 Dir 0 Deleted Dir: /folder
0 dir29 Dir 0 1 dir33 Dir 0
0 dir30 Dir 0 1 dir34 Dir 0
0 dir31 Dir 0 1 dir35 Dir 0
0 dir32 Dir 0
0 dir33 Dir 0 Deleted Dir: /folder/dir10
0 dir34 Dir 0 Deleted Dir: /folder/dir11
0 dir35 Dir 0 Deleted Dir: /folder/dir12

```

Test 6

The Conclusion

The implementation in its current state would not be suitable for larger disks due not having a quick directory reference, but for our 1024 KiB disk it is quite fine. With the load/save functions, it could even work as a custom floppy-disk file system.

Overall, the assignment shows that it doesn't take that much to create a simple file system. Or even a more complex one. Adding journaling, write buffering, and a read-ahead cache to this implementation would be fairly straight-forward.

Source Listing 1: misc.cpp

```

#include "disk.h"
#include "filesystem.h"

// Loads the map of free space
void AllocateFMap() {
    mapsize = (int)ceil(((double)DISK_SIZE) / (((double)SECTOR_SIZE)*8.0));
    freemap.reset();

    char *buf = new char[SECTOR_SIZE];
    for(int i=0;i<mapsize;i++) {
        memset(buf, 0, SECTOR_SIZE);
        read_sector(RESERVED+i, buf);
        for(int j=0;j<SECTOR_SIZE;j++) {
            if (j*8+i*SECTOR_SIZE*8 >= DISK_SIZE)
                break;
            if (buf[j] & 1)
                freemap[j*8+i*SECTOR_SIZE*8] = true;
            if (buf[j] & 2)
                freemap[j*8+1+i*SECTOR_SIZE*8] = true;
            if (buf[j] & 4)
                freemap[j*8+2+i*SECTOR_SIZE*8] = true;
            if (buf[j] & 8)
                freemap[j*8+3+i*SECTOR_SIZE*8] = true;
            if (buf[j] & 16)
                freemap[j*8+4+i*SECTOR_SIZE*8] = true;
            if (buf[j] & 32)
                freemap[j*8+5+i*SECTOR_SIZE*8] = true;
            if (buf[j] & 64)
                freemap[j*8+6+i*SECTOR_SIZE*8] = true;
            if (buf[j] & 128)
                freemap[j*8+7+i*SECTOR_SIZE*8] = true;
        }
    }
    // Make sure the first RESERVED+mapsize sectors are marked as used
    for(int i=0;i<RESERVED+mapsize;i++) {
        freemap[i] = true;
    }
    /*
    for(int i=0;i<freemap.size();i++) {
        cout << (freemap.at(i) ? "1" : "0");
    }
    cout << endl;
    */
}

// Writes the free space map
void UpdateFMap() {
    char *buf = new char[SECTOR_SIZE];
    for(int i=0;i<mapsize;i++) {
        memset(buf, 0, SECTOR_SIZE);
        for(int j=0;j<SECTOR_SIZE;j++) {
            if (j*8+i*SECTOR_SIZE*8 >= DISK_SIZE)
                break;
            if (freemap[j*8+i*SECTOR_SIZE*8] == true)
                buf[j] |= 1;
            if (freemap[j*8+1+i*SECTOR_SIZE*8] == true)
                buf[j] |= 2;
            if (freemap[j*8+2+i*SECTOR_SIZE*8] == true)
                buf[j] |= 4;
            if (freemap[j*8+3+i*SECTOR_SIZE*8] == true)
                buf[j] |= 8;
            if (freemap[j*8+4+i*SECTOR_SIZE*8] == true)
                buf[j] |= 16;
            if (freemap[j*8+5+i*SECTOR_SIZE*8] == true)
                buf[j] |= 32;
            if (freemap[j*8+6+i*SECTOR_SIZE*8] == true)
                buf[j] |= 64;
            if (freemap[j*8+7+i*SECTOR_SIZE*8] == true)
                buf[j] |= 128;
        }
        write_sector(RESERVED+i, buf);
    }
}

// Searches for a spot with 'want' sequentially free sectors
int SearchFMap(unsigned int want) {
    if (want < DISK_SIZE-(RESERVED+mapsize)) {
        // No need to search in the first RESERVED+mapsize sectors.
        for(int i=RESERVED+mapsize;i<freemap.size()-want;i++) {
            if (freemap[i] == false) {
                unsigned int found = 0; // Assume this spot is good
            }
        }
    }
}

```

```

        for(int j=0;j<want;j++) {
            if (freemap[j+i] == false) {
                found++;
            } else
                break; // Spot proven bad
        }
        if (found == want)
            return i;
    }
}

}
fprintf(flog,"Could not allocate %u sequential sectors. Total available: %u\n",
        want, DISK_SIZE-freemap.count());
return ERR_NOSPACE;
}

// Loads FAT into memory
void AllocateFAT() {
    numEntries = (int)floor((((double)RESERVED)*
        ((double)SECTOR_SIZE)/((double)sizeof(FSEntry)));
    if (entries != NULL)
        delete(entries);
    entries = new FSEntry[numEntries];

    char *ptr = (char *)entries;
    for(int i=0;i<RESERVED;i++) {
        read_sector(i, &ptr[i*SECTOR_SIZE]);
    }

    for(int i=0;i<numEntries;i++) {
        if (entries[i].flags & FLAG_DELETED
            || entries[i].flags == 0
            || entries[i].start == 0)
            continue;
        for(int j=entries[i].start ; j<=entries[i].stop ; j++)
            freemap[j] = true;
    }
}

// Writes the entire FAT to disk
void UpdateFAT() {
    char *ptr = (char *)entries;
    for(int i=0;i<RESERVED;i++) {
        write_sector(i, &ptr[i*SECTOR_SIZE]);
    }
}

// Writes the FAT atomically
int AtomicUpdateFAT(unsigned int e) {
    if (e >= numEntries)
        return ERR_BADFAT; // Out of bounds

    // Make sure to mark or unmark free space
    if (!(entries[e].flags & FLAG_DELETED) && (entries[e].flags != 0)
        && (entries[e].start != 0)) {
        for(int i=entries[e].start ; i<=entries[e].stop && i<DISK_SIZE ; i++)
            freemap[i] = true;
    } else if ((entries[e].flags & FLAG_DELETED) && (entries[e].start != 0)) {
        for(int i=entries[e].start ; i<=entries[e].stop && i<DISK_SIZE ; i++)
            freemap[i] = false;
    }

    // Calculate which sector has to be updated
    double w = ((double)e) * ((double)sizeof(FSEntry)) / ((double)SECTOR_SIZE);
    int s = (int)floor(w);
    char *ptr = (char *)entries;
    write_sector(s, &ptr[s*SECTOR_SIZE]);

    // Determine if the FSEntry was stored across multiple sectors
    // This is not optimal for such small sectors, but just to show
    // what could be done if you really had to minimize disk access.
    if (w*SECTOR_SIZE + sizeof(FSEntry) > ceil(w)*SECTOR_SIZE) {
        s++;
        if (s < numEntries)
            write_sector(s, &ptr[s*SECTOR_SIZE]);
    }

    return 0;
}

void disk_wipe() {
    char *nul = new char[SECTOR_SIZE];
    memset(nul, 0, SECTOR_SIZE);
}

```

```
    for (int i=0;i<DISK_SIZE;i++)
        write_sector(i, nul);
}

void disk_save() {
    char *buf = new char[SECTOR_SIZE];
    FILE *fd = fopen("diskimage.img", "wb");
    for(int i=0;i<DISK_SIZE;i++) {
        memset(buf, 0, SECTOR_SIZE);
        read_sector(i, buf);
        fwrite(buf, 1, SECTOR_SIZE, fd);
    }
    fclose(fd);
}

void disk_load() {
    char *buf = new char[SECTOR_SIZE];
    FILE *fd = fopen("diskimage.img", "rb");
    for(int i=0;i<DISK_SIZE;i++) {
        memset(buf, 0, SECTOR_SIZE);
        fread(buf, 1, SECTOR_SIZE, fd);
        write_sector(i, buf);
    }
    fclose(fd);
}

// Splits a string into it's component parts, delimited by a char.
// Used in this case to split by / for the paths.
vector<string> explode(const char by, string splitme) {
    vector<string> hubba;
    int pos=string::npos, cur=-1;
    splitme.append(&by);
    while( (pos = splitme.find(by, cur+1)) != string::npos ) {
        if (pos-cur-1 > 0)
            hubba.push_back(splitme.substr(cur+1, pos-cur-1));
        cur=pos;
    }

    return hubba;
}
```

Source Listing 2: filesystem.cpp

```

#include "disk.h"
#include "filesystem.h"

unsigned int numEntries = 0;
FSEntry *entries = NULL;
File *files = NULL;
bitset<DISK_SIZE> freemap;
unsigned int mapsize = 0;
FILE *flog = NULL;

int dm14_format() {
    int i;

    // In order to format, just mark all files as deleted.
    // This is fairly consistent with how real file systems handle formatting,
    // since some allow for unformatting again.
    // Some also do a surface integrity check, but I don't have a surface.
    for(i=1;i<numEntries;i++) {
        entries[i].flags |= FLAG_DELETED;
    }
    UpdateFAT(); // Write entire FAT to disk
    UpdateFMap(); // Unmark used space

    return 0; // Cannot fail...
}

// Makes use of C++ to greatly simplify parsing the path.
int GetLastParent(char *path) {
    int cur=0;
    vector<string> names = explode('/', string(path));
    vector<string>::iterator it;

    if (names.size() == 0)
        return 0; // Special case, root dir.

    for(it = names.begin() ; it != names.end() ; it++) {
        string bit = *it;

        if ((bit.length() < 1) || (bit.length() > 15))
            return ERR_BADPATH; // Name too long or too short

        bool found = false;
        for(int i=1;i<numEntries;i++) {
            if (
                (entries[i].parent == cur) // Only consider elements in the current path
                && (entries[i].flags & FLAG_DIR) // Only consider dirs
                && (strcmp(entries[i].name, bit.c_str()) == 0)
            ) {
                cur = i;
                found = true;
                break;
            }
        }
        if (!found)
            return ERR_BADPATH; // Invalid path
    }
    return cur; // Return last direct parent
}

int dm14_delete(char *path) {
    int i,c;

    char fname[16], *tname = NULL;
    memset(fname, 0, 16);

    path = strdup(path);
    if ((tname = strrchr(path, '/')) == NULL)
        return ERR_BADPATH;
    tname = (char *)&tname[1]; // Skip the leading /
    if ((strlen(tname) < 1) || (strlen(tname) > 15))
        return ERR_BADNAME; // Too short or too long.

    strcpy(fname, tname); // Save for later
    tname[-1] = '\0'; // Remove filename from path

    c = GetLastParent(path);
    if (c >= 0) {
        for(i=0;i<numEntries;i++) {
            if ((entries[i].parent != c)
                || (entries[i].flags == 0)
                || (entries[i].flags & FLAG_DELETED))

```

```

        continue;
    if (strcmp(entries[i].name, fname) == 0) {
        if (entries[i].flags & FLAG_DIR) {
            for(int j=0;j<numEntries;j++) {
                if (entries[j].parent == i) {
                    char *tmp = new char[1024];
                    sprintf(tmp, "%s/%s/%s", path, fname, entries[j].name);
                    dm14_delete(tmp); // Recursive deletion
                }
            }
        }
        entries[i].flags |= FLAG_DELETED;
        AtomicUpdateFAT(i);
        fprintf(flog, "Deleted %s: %s/%s\n",
            ((entries[i].flags & FLAG_DIR) ? "Dir" : "File"),
            path, fname);
    }
}
return c;
}

int dm14_ls(char *path) {
    int i,c;

    c = GetLastParent(path);
    if (c >= 0) {
        fprintf(flog, "Listing of %s\n", path);
        for(i=0;i<numEntries;i++) {
            if ((entries[i].parent != c)
                || (entries[i].flags == 0)
                || (entries[i].flags & FLAG_DELETED))
                continue;
            fprintf(flog, "%u\t%s\t%s\t%u\n", c, entries[i].name,
                ((entries[i].flags & FLAG_DIR) ? "Dir" : "File"),
                entries[i].length);
        }
        return 0;
    }
    return c; // Invalid path
}

int dm14_seek(unsigned int fd, unsigned int pos) {
    if ((fd == 0) || (fd > MAX_FILES))
        return ERR_BADFD; // Bad file descriptor
    fd--;

    files[fd].pos = pos;

    return 0;
}

int dm14_mkdir(char *path, char *name) {
    int i,c,e;

    c = GetLastParent(path);
    if (c >= 0) {
        for(i=1;i<numEntries;i++) {
            // Locate an unused FSEntry
            if ((entries[i].flags == 0)
                || (entries[i].flags & FLAG_DELETED)) {
                entries[i].flags = FLAG_DIR;
                entries[i].length = 0;
                strcpy(entries[i].name, name);
                entries[i].parent = c;
                entries[i].start = entries[i].stop = 0;
                if ((e = AtomicUpdateFAT(i)) != 0)
                    return e;
                return 0;
            }
        }
        return ERR_FULLFAT; // No available FSEntry
    }
    return c; // Invalid path
}

int dm14_mkfile(char *path, char *name) {
    int i,c,e;

    c = GetLastParent(path);
    if (c >= 0) {
        for(i=1;i<numEntries;i++) {
            // Locate an unused FSEntry
            if ((entries[i].flags == 0) || (entries[i].flags & FLAG_DELETED)) {

```

```

        entries[i].flags = FLAG_FILE;
        entries[i].length = 0;
        strcpy(entries[i].name, name);
        entries[i].parent = c;
        entries[i].start = entries[i].stop = 0;
        if ((e = AtomicUpdateFAT(i)) != 0)
            return e;
        return i;
    }
    }
    return ERR_FULLFAT; // No available FSEntry
}
return c; // Invalid path
}

int dm14_open(char *path) {
    int i,j,c;
    char fname[16], *tname = NULL;
    memset(fname, 0, 16);

    path = strdup(path);
    if ((tname = strrchr(path, '/')) == NULL)
        return ERR_BADPATH;
    tname = (char *)&tname[1]; // Skip the leading /
    if ((strlen(tname) < 1) || (strlen(tname) > 15))
        return ERR_BADNAME; // Too short or too long.

    strcpy(fname, tname); // Save for later
    tname[0] = '\0'; // Remove filename from path

    c = GetLastParent(path);
    if (c >= 0) {
        // Assume the file exists and search for it.
        for(i=1;i<numEntries;i++) {
            if ((entries[i].parent != c) || (entries[i].flags == 0)
                || (entries[i].flags & FLAG_DELETED) || (entries[i].flags & FLAG_DIR))
                continue;
            if (strcmp(entries[i].name, fname) == 0) {
                for(j=0;j<MAX_FILES;j++) {
                    if (files[j].file == 0) {
                        files[j].file = i;
                        files[j].pos = 0;
                        return j+1; // As to not confuse it with 0, the non-error value.
                    }
                }
                return ERR_FDFULL; // No available file descriptors
            }
        }

        // Right, the file does not exist...create it.
        i = dm14_mkfile(path, fname); // 'touch' the file into existence.
        if (i >= 0) {
            for(j=0;j<MAX_FILES;j++) {
                if (files[j].file == 0) {
                    files[j].file = i;
                    files[j].pos = 0;
                    return j+1; // As to not confuse it with 0, the non-error value.
                }
            }
            return ERR_FDFULL; // No available file descriptors
        }
        return i; // Invalid path, or no available FSEntry
    }
    return c; // Invalid path
}

int dm14_write(unsigned int fd, char *data, unsigned int length) {
    if ((fd == 0) || (fd > MAX_FILES))
        return ERR_BADFD; // Bad file descriptor
    fd--;

    fprintf(flog,"Will write %u byte to fd %u\n", length, fd);

    if (length == 0)
        return 0;

    FSEntry *ent = &entries[files[fd].file];
    fprintf(flog,"Writing to entry %u at pos %u\n", files[fd].file, files[fd].pos);
    fprintf(flog,"Filename %s with parent %u\n", ent->name, ent->parent);

    // If this file has never been written to before, allocate space for it now.
    if (ent->start == 0) {
        int h,x=2,n=(int)ceil(((double)(length+files[fd].pos))/((double)SECTOR_SIZE));
        // Try to allocate needed size+2 sectors.
    }
}

```

```

    if ((h = SearchFMap(n+x)) <= 0) {
        // Ok, no space for that...so try +1 only.
        x--;
        if ((h = SearchFMap(n+x)) <= 0) {
            // No space for that either. Try getting only what we need.
            x--;
            if ((h = SearchFMap(n)) <= 0) {
                return h; // Definetly out of space
            }
        }
    }
    ent->start = h;
    ent->stop = h+n+x;
}

fprintf(flog,"Start sector %u, stop sector %u\n", ent->start, ent->stop);

// Make sure the file length is updated
if (files[fd].pos > ent->length)
    ent->length = files[fd].pos;

// Write...
char *buf = new char[SECTOR_SIZE];
int datapos = 0;
while(length > 0) {
    memset(buf, 0, SECTOR_SIZE);

    // Figure out which sector, relative to ent->start, we have to access for this write
    int c = (int)floor(((double)files[fd].pos)/((double)SECTOR_SIZE));
    if (c+ent->start > ent->stop)
        return ERR_EOF;
    if (c+ent->start >= DISK_SIZE)
        return ERR_EOD;
    read_sector(c+ent->start, buf);

    // Copy to the currently loaded sector
    int bufpos = files[fd].pos - c*SECTOR_SIZE;
    int wlen = SECTOR_SIZE - bufpos;
    if (wlen > length)
        wlen = length;
    memcpy(&buf[bufpos], &data[bufpos+datapos], wlen);

    // Write the sector, and update vars
    write_sector(c+ent->start, buf);
    datapos += wlen;
    files[fd].pos += wlen;
    ent->length += wlen;
    length -= wlen;
    fprintf(flog,"Wrote %u byte to sector %u\n", wlen, c+ent->start);
}

AtomicUpdateFAT(files[fd].file);
fprintf(flog,"\n");
return 0;
}

int dm14_close(unsigned int fd) {
    if ((fd == 0) || (fd > MAX_FILES))
        return ERR_BADFD; // Bad file descriptor
    fd--;

    files[fd].file = 0;
    files[fd].pos = 0;

    return 0;
}

int dm14_read(unsigned int fd, char *data, unsigned int length) {
    if ((fd == 0) || (fd > MAX_FILES))
        return ERR_BADFD; // Bad file descriptor
    fd--;

    fprintf(flog,"Will read %u byte from fd %u\n", length, fd);

    if (length == 0)
        return 0;

    FSEntry *ent = &entries[files[fd].file];
    if (files[fd].pos+length > ent->length)
        return ERR_EOF; // Can't read beyond EOF

    fprintf(flog,"Reading from entry %u at pos %u\n", files[fd].file, files[fd].pos);
    fprintf(flog,"Filename %s with parent %u\n", ent->name, ent->parent);
    fprintf(flog,"Start sector %u, stop sector %u\n", ent->start, ent->stop);
}

```

```

// Read...
char *buf = new char[SECTOR_SIZE];
int datapos = 0;
while(length > 0) {
    memset(buf, 0, SECTOR_SIZE);

    // Figure out which sector, relative to ent->start, we have to access for this read
    int c = (int)floor(((double)files[fd].pos)/((double)SECTOR_SIZE));
    if (c+ent->start > ent->stop)
        return ERR_EOF;
    read_sector(c+ent->start, buf);

    // Copy to data
    int bufpos = files[fd].pos - c*SECTOR_SIZE;
    int rlen = SECTOR_SIZE - bufpos;
    if (rlen > length)
        rlen = length;
    memcpy(&data[bufpos+datapos], &buf[bufpos], rlen);

    // Update vars
    datapos += rlen;
    files[fd].pos += rlen;
    length -= rlen;
    fprintf(flog,"Read %u byte\n", rlen);
}

fprintf(flog,"\n");
return 0;
}

#define M_PI          3.14159265358979323846
int main() {
    flog = fopen("out.log", "wb");

    files = new File[sizeof(File)*MAX_FILES];
    memset(files, 0, sizeof(File)*MAX_FILES);

    disk_wipe();
    //disk_load();
    AllocateFMap();
    AllocateFAT();

    fprintf(flog, "Disk has %u FAT entries, of %u byte each.\n", numEntries, sizeof(FSEntry));

    dm14_mkdir("/", "folder");
    dm14_mkdir("/folder/", "subfolder");
    dm14_mkdir("/folder/subfolder/", "subsub");

    int fd = dm14_open("/folder/subfolder/file");
    if (fd > 0) {
        int sz = (int)(SECTOR_SIZE*M_PI);
        char *buf = new char[sz];
        for(int i=0;i<sz;i++)
            buf[i] = (i%0xFF);
        dm14_seek(fd, 67);
        int e = dm14_write(fd, buf, sz);
        if (e < 0)
            fprintf(flog,"Error: %i\n", e);
        e = dm14_write(fd, buf, sz/2);
        if (e < 0)
            fprintf(flog,"Error: %i\n", e);
        dm14_close(fd);
    } else
        fprintf(flog,"Error: %i\n", fd);

    fd = dm14_open("/folder/file");
    if (fd > 0) {
        char buf[] = "Hello World";
        dm14_seek(fd, SECTOR_SIZE-5);
        int e = dm14_write(fd, buf, sizeof(buf));
        if (e < 0)
            fprintf(flog,"Error: %i\n", e);
        dm14_seek(fd, 0);
        e = dm14_write(fd, buf, sizeof(buf));
        if (e < 0)
            fprintf(flog,"Error: %i\n", e);

        char *vuf = new char[128];
        dm14_seek(fd, 0);
        e = dm14_read(fd, vuf, sizeof(buf));
        if (e < 0)
            fprintf(flog,"Error: %i\n", e);
        fprintf(flog,"Read '%s'\n", vuf);
    }
}

```

```

    dm14_seek(fd, SECTOR_SIZE);
    e = dm14_read(fd, vuf, sizeof(buf)-5);
    if (e < 0)
        fprintf(flog,"Error: %i\n", e);
    fprintf(flog,"Read '%s'\n", vuf);
    dm14_close(fd);
} else
    fprintf(flog,"Error: %i\n", fd);

fd = dm14_open("/folder/file");
if (fd > 0) {
    int sz = (int)(SECTOR_SIZE*M_PI/4);
    char *buf = new char[sz];
    for(int i=0;i<sz;i++)
        buf[i] = (i%0xFF);
    int e = 0;
    while (e==0) {
        e = dm14_write(fd, buf, sz);
        if (e < 0)
            fprintf(flog,"Error: %i\n", e);
    }
    dm14_close(fd);
} else
    fprintf(flog,"Error: %i\n", fd);

fd = dm14_open("/bigfile");
if (fd > 0) {
    int sz = (int)(SECTOR_SIZE*(DISK_SIZE-RESERVED-14));
    char *buf = new char[sz];
    for(int i=0;i<sz;i++)
        buf[i] = (i%0xFF);
    int e = 0;
    while (e==0) {
        e = dm14_write(fd, buf, sz);
        if (e < 0)
            fprintf(flog,"Error: %i\n", e);
    }
    dm14_close(fd);
} else
    fprintf(flog,"Error: %i\n", fd);

dm14_ls("/");
dm14_ls("/folder/");
dm14_ls("/folder/subfolder/");
dm14_ls("/folder/subfolder/subdir/");

dm14_delete("/folder");

int i,e;
// Saturate the available file descriptor
for(e=0,i=0;e>=0;i++) {
    e = dm14_open("/bigfile");
}
fprintf(flog, "Saturated descriptors after %u open. Error %i\n", i, e);

// Fill the file system with entries
char *n = new char[16];
for(e=0,i=10;e==0;i++) {
    sprintf(n, "dir%u", i);
    e = dm14_mkdir("/", n);
}
dm14_ls("/");
dm14_format(); // Wipe them all the quick way
dm14_mkdir("/", "folder"); // Subdir for easier deletion
for(e=0,i=10;e==0;i++) {
    sprintf(n, "dir%u", i);
    e = dm14_mkdir("/folder/", n);
}
// List and delete them all
dm14_ls("/folder/");
dm14_delete("/folder");

UpdateFAT();
UpdateFMap();
disk_save();

return 0;
}

```

Source Listing 3: filesystem.h

```
#ifndef DM14_FS_H
#define DM14_FS_H

#define ERR_BADPATH          -1
#define ERR_FULLFAT         -2
#define ERR_FULLDISK        -3
#define ERR_BADNAME         -4
#define ERR_NOSUCH          -5
#define ERR_BADFD           -6
#define ERR_BADSEEK         -7
#define ERR_FDFULL          -8
#define ERR_BADFAT          -9
#define ERR_NOSPACE         -10
#define ERR_EOF              -11
#define ERR_EOD              -12

#define FLAG_FILE            1
#define FLAG_DIR             2
#define FLAG_DELETED        4

// This amount of sectors are reserved for the FAT
#define RESERVED            2

typedef struct _FSEntry {
    unsigned char flags;
    unsigned int start, stop; // Sector endpoints of the file
    unsigned int length;
    unsigned int parent;
    char name[16];
} FSEntry;

#define MAX_FILES 16
typedef struct _File {
    unsigned int file; // FSEntry
    unsigned int pos; // Internal seek position
} File;

extern unsigned int numEntries;
extern FSEntry *entries;
extern File *files;
extern bitset<DISK_SIZE> freemap;
extern unsigned int mapsize;
extern FILE *flog;

void disk_wipe();
void disk_save();
void disk_load();
void AllocateFAT();
void UpdateFAT();
int AtomicUpdateFAT(unsigned int e);
void AllocateFMap();
void UpdateFMap();
int SearchFMap(unsigned int want);
vector<string> explode(const char by, string splitme);

#endif
```