**Computer Science 360**
**Assignment 3**
**Due: March 28, 2016 at 11:59pm**

# Introduction

In this assignment you will implement utilities that perform operations on a file system similar to Microsoft's FAT file system.

# Sample File Systems

You have been given four file system images: `disk1.img`, `disk1_empty.img`, `disk2.img` and `disk2_empty.img`.

You should get comfortable examining the raw data in the file system images using the program `xxd`.

# Part I – 5 marks

In part I, you will write a program that displays information about the file system. In order to complete part I, you will need to read the file system super block and use the information in the super block to read in the FAT.

Your program for part I will be invoked as follows:

```
./diskinfo disk1.img
```

Sample output:

```
Super block information:
Block size: 512
Block count: 15360
FAT starts: 1
FAT blocks: 120
Root directory start: 121
Root directory blocks: 4

FAT information:
Free Blocks: 15235
Reserved Blocks: 121
Allocated Blocks: 4
```

Please be sure to use the exact same output format as shown above.

## Part II – 5 marks

In part II, you will write a program that displays the contents of the root directory in the file system.

Your program for part II will be invoked as follows:

```
./disklist disk1.img
```

The first column will contain an F for regular files and a D for directories, followed by a single space. Next you will use 10 characters to show the file size, followed by a single space and then 30 characters for the file name, followed by a single space, followed by the file modification date (we won't display the file creation date).

For example:

```
F          91                          readme.txt 2016/03/02 12:20:19
F          56                             bar.txt 2016/03/02 12:20:43
F       25600                           file3.txt 2016/03/02 12:20:51
F       51200                           file4.txt 2016/03/02 12:20:55
F       38512                              ls_mac 2016/03/02 12:21:42
```

## Part III – 5 Marks

In part III you will write a program that copies a file from the file system to the current directory in Unix. If the specified file is not found in the root directory of the file system, you should output the message “File not found.” on a single line and exit.

Your program for part III will be invoked as follows:

```
./diskget disk1.img foo.txt
```

## Part IV – 5 marks

In part IV you will write a program that copies a file from the current Unix directory into the file system. If the specified file is not found, you should output the message “File not found.” on a single line and exit.

Your program for part IV will be invoked as follows:

```
./diskput disk1.img foo.txt
```

## Accessing the file system

Unix allows you to treat devices as if they were files, so we can use the stdio functions to manipulate our file system.

You will want to become familiar with the library functions: `fopen, fseek, fread` and `fwrite`.

## What to hand in

You need to hand in a .tar file containing all your source code and a Makefile that produces the excutables for parts 1 – 4.

Please include a readme.txt file that explains any bonus activities that you completed.

## Bonus

There is lots of room for bonus marks in this assignment. As before, you must get permission from your instructor before you begin the bonus activities. Some things you may consider:

- implementing directories other than the root directory
- implementing fast searching for directories and/or free space
- implementing the file system as a device driver for Linux (!hard!)
- writing a shell to interact with the filesystem

If you have any other ideas, please email your instructor.

## Grading

The four programs created have equal weight. Each part is worth 10 points. We will test your implementation against the four disk images we have supplied as well as an additional disk image that meets the specifications given in this assignment.

# File System Specification

The file system has 3 major components:  the super block, the File Allocation Table (FAT) and the directory structure.

Each of these 3 components is describe in the sections below.

# File system Superblock

The first block (512 bytes) is reserved to contain information about the file system.  The layout of the superblock is as follows:

| ***Description*** | ***Size*** | ***Default Value*** |
|---|---|---|
| File system identifier | 8 bytes | CSC360FS |
| Block Size | 2 bytes | 0x200 |
| File system size (in blocks) | 4 bytes | 0x00001400 |
| Block where FAT starts | 4 bytes | 0x00000001 |
| Number of blocks in FAT | 4 bytes | 0x00000028 |
| Block where root directory starts | 4 bytes | 0x00000029 |
| Number of blocks in root dir | 4 bytes | 0x00000008 |

# Directory Entries

Each directory entry takes up 64 bytes, which implies there are 8 directory entries per 512 byte block.

Each directory entry has the following structure:

| ***Description*** | ***Size*** |
|---|---|
| Status | 1 byte |
| Starting Block | 4 bytes |
| Number of Blocks | 4 bytes |
| File Size (in bytes) | 4 bytes |
| Create Time | 7 bytes |
| Modify Time | 7 bytes |
| File Name | 31 bytes |
| unused (set to 0xFF) | 6 bytes |

### *Status*

This is bit mask that is used to describe the status of the file. Currently only 3 of the bits are used.

Bit 0 - set to 0 if this directory entry is available, set to 1 if it is in use
Bit 1 – set to 1 if this entry is a normal file
Bit 2 – set to 1 if this entry is a directory

It is implied that only one of bit 2 or bit 1 can be set to 1. That is, an entry is either a normal file or it is a directory, not both.

### *Starting Block*

This is the location on disk of the first block in the file

### *Number of Blocks*

The total number of blocks in this file

### *File Size*

The size of the file, in bytes. The size of this field implies that the largest file we can support is 2^32 bytes long.

### *Create Time*

The date and time when this file was created.

The file system stores the times as integer values in the format:

YYYYMMDDHHMMSS

2 bytes used to store YYYY
1 byte to store MM
1 byte to store DD
1 byte to store HH
1 byte to store MM
1 byte to store SS

### *Modify Time*

The last time this file was modified. Stored in the same format as the Create Time shown above.

### *File Name*

The file name, null terminated. Because of the null terminator, the maximum length of any filename is 30 bytes.

Valid characters are upper and lower case letters (a-z, A-Z), digits (0-9) and the underscore character (_).

## File Allocation Table (FAT)

Each directory entry contains the starting block number for a file, lets say it is block X. To find the next block in the file, you should look at entry X in the FAT. If the value you find there does not indicate End-of-File (see below) then that value, call it Y, is the next block in the file.

That is, the first block is block X, you look in the FAT table at position X and find the value Y. The second data block is block Y. Then you look in the FAT at position Y to find the next data block... continue this until you find the special value in the FAT indicating that you are at the end of the file.

The FAT is really just a linked list, which the head of the list being stored in the directory entry, and the 'next pointers' being stored in the FAT.

Fat entries are 4 bytes long (32 bits), which implies there are 128 FAT entries per block.

Special values for FAT entries are:

| ***Value*** | ***Meaning*** |
|---|---|
| 0x00000000 | This block is available |
| 0x00000001 | This block is reserved |
| 0x00000002 – 0xFFFFFF00 | Allocated blocks as part of files |
| 0xFFFFFFFF | This is the last block in a file |

## Byte Ordering

Different hardware architectures store multi-byte data (like integers) in different orders. Consider the large integer: `0xDEADBEEF`

On the Intel architecture (Little Endian), it would be stored in memory as:

```
EF BE AD DE
```

On the PowerPC (Big Endian), it would be stored in memory as:

```
DE AD BE EF
```

Our file system will use Big Endian for storage. This will make debugging the file system by examining the raw data much easier.

This will mean that you have to convert all your integer values to Big Endian before writing them to disk. There are utility functions in `netinit/in.h` that do exactly that. (When sending data over the network, it is expected the data is in Big Endian format.)

See the functions `htons, htonl, ntohs` and `ntohl`.

The side effect of using these functions will be that your code will work on multiple platforms. (On machines that natively store integers in Big Endian format the above functions don't actually do anything – but you should still use them!)