CSc 360: Operating Systems (Fall 2023)

Programming Assignment 3
P3: A Simple File System (SFS)
Spec Out: Oct 30, 2023
Code Due: Nov 27, 2023

1 Introduction

So far, you have built a shell environment and a multi-thread scheduler with process synchronization. Excellent job! What is still missing for a "real" operating system? A file system! In this assignment, you will implement utilities that perform operations on a file system similar to Microsoft's FAT file system with some improvement.

1.1 Sample File Systems

You will be given a test file system image for self-testing, but you can create your own image following the specification, and your submission may be tested against other disk images following the same specification.

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

IMPORTANT: since you are dealing with binary data, functions intended for string manipulation such as `strcpy()` do NOT work (since binary data may contain binary '0' anywhere), and you should use functions intended for binary data such as `memcpy()`.

2 Tutorial Schedule

In order to help you finish this programming assignment on time successfully, the schedule of the lectures has been updated to synchronize with the tutorials and the assignment. There are three tutorials arranged during the course of this assignment. **NOTE**: Please do attend the tutorials and follow the tutorial schedule closely.

Date	**Tutorial**	**Milestones**
Oct 31/Nov 1/3	P3 spec go-thru and practice questions	design done
Nov 7/8/10	more on design and implementation	diskinfo/disklist done
Nov 14/15/17	Reading break, **no tutorials this week**	diskget/diskput done
Nov 21/22/24	testing and submission instructions	final deliverable

3 Requirements

3.1 Part I (3 points)

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 the FAT.

Your program for Part I will be invoked as follows (output values here are just for illustration purposes):

```
./diskinfo test.img
```

Sample output:

```
Super block information
Block size: 512
Block count: 5120
FAT starts: 1
FAT blocks: 40
Root directory starts: 41
Root directory blocks: 8

FAT information
Free blocks: 5071
Reserved blocks: 41
Allocated blocks: 8
```

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

3.2 Part II (4 points)

In Part II, you will write a program, with the routines already implemented for Part I, that displays the contents of the root directory or a given sub-directory in the file system.

Your program for Part II will be invoked as follows:

```
./disklist test.img /sub_dir
```

The directory listing should be **formatted** as follows:

1. The first column will contain:
   (a) F for regular files, or
   (b) D for directories;
   followed by a single space
2. then 10 characters to show the file size, followed by a single space
3. then 30 characters for the file name, followed by a single space
4. then the file creation date and time

For example:

```
F       2560                foo.txt 2015/11/15 12:00:00
F       5120               foo2.txt 2015/11/15 12:00:00
F      48127                 makefs 2015/11/15 12:00:00
F          8               foo3.txt 2015/11/15 12:00:00
```

3.3 Part III (4 points)

In Part III, you will write a program that copies a file from the file system to the current directory in Linux. If the specified file is not found in the root directory or a given sub-directory of the file system, you should output the message

```
File not found.
```

and exit.

Your program for Part III will be invoked as follows:

```
./diskget test.img /sub_dir/foo2.txt foo.txt
```

3.4 Part IV (4 points)

In Part IV, you will write a program that copies a file from the current Linux directory into the file system, at the root directory or a given sub-directory. 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 test.img foo.txt /sub_dir/foo3.txt
```

4 File System Specification

The FAT file system has three major components:

1. the super block,
2. the File Allocation Table (informally referred to as the FAT),
3. the directory structure.

Each of these three components is described in the subsections below.

4.1 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

Note: Block number starts from 0 in the file system.

4.2 Directory Entries

Each directory entry takes 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
Creation Time	7 bytes
Modification Time	7 bytes
File Name	31 bytes
unused (set to 0xFF)	6 bytes

The description of each field is as follows:

Status This is a bit mask that is used to describe the status of the file. Currently only 3 of the bits are used. It is implied that only one of bit 2 or bit 1 can be set to1. That is, an entry is either a normal file or it is a directory, *not both*.

```
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
```

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.
Creation Time The date and time when this file was created. The file system stores the system times as integer values in the format: YYYYMMDDHHMMSS

Field	Size
YYYY	2 bytes
MM	1 byte
DD	1 byte
HH	1 byte
MM	1 byte
SS	1 byte

Modification Time The last time this file was modified. Stored in the same format as the Creation 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 (`_`).

4.3 File Allocation Table (FAT)

Each directory entry contains the starting block number for a file, let's say it is block number `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 (i.e., the last block, see below) then that value, say `Y`, is the next block number in the file.

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

The FAT is really just a linked list, with the head of the list being stored in the "Starting Block" field in the directory entry, and the 'next pointers' being stored in the FAT entries.

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

Special values for FAT entries are described in the following.

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`

5 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 too.)

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, like the Mac (not the ARM or Intel-based ones), the above functions don't actually do anything but you should still use them!)

6 Submission Requirements

What to hand in: You need to hand in a `.tar.gz` file containing all your source code and a Makefile that produces the executables for Parts I–IV.

Please include a `readme.txt` file that explains your design and implementation.

The file is submitted through `bright.uvic.ca` site.

A An Exercise

Q1 Consider the superblock shown below:

```
0000000: 4353 4333 3630 4653 0200 0000 1400 0000  CSC360FS........
0000010: 0001 0000 0028 0000 0029 0000 0008 0000  .....(...).....
0000020: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

(a) Which block does the FAT start from? How many blocks are used for the FAT?

(b) Which block does the root directory start from? How many blocks are used for the root directory?

Q2 Consider the following block from the root directory:

```
0005200: 0300 0000 3100 0000 0500 000a 0007 d50b ....1...........
0005210: 0f0c 0000 07d5 0b0f 0c00 0066 6f6f 2e74 ...........foo.t
0005220: 7874 0000 0000 0000 0000 0000 0000 0000 xt..............
0005230: 0000 0000 0000 0000 0000 00ff ffff ffff ................
0005240: 0300 0000 3600 0000 0a00 0014 0007 d50b ....6...........
0005250: 0f0c 0000 07d5 0b0f 0c00 0066 6f6f 322e ...........foo2.
0005260: 7478 7400 0000 0000 0000 0000 0000 0000 txt.............
0005270: 0000 0000 0000 0000 0000 00ff ffff ffff ................
0005280: 0300 0000 4000 0000 5e00 00bb ff07 d50b ....@...^.......
0005290: 0f0c 0000 07d5 0b0f 0c00 006d 616b 6566 ...........makef
00052a0: 7300 0000 0000 0000 0000 0000 0000 0000 s...............
00052b0: 0000 0000 0000 0000 0000 00ff ffff ffff ................
00052c0: 0300 0000 9e00 0000 0100 0000 0807 d50b ................
00052d0: 0f0c 0000 07d5 0b0f 0c00 0066 6f6f 332e ...........foo3.
00052e0: 7478 7400 0000 0000 0000 0000 0000 0000 txt.............
00052f0: 0000 0000 0000 0000 0000 00ff ffff ffff ................
```

(a) How many files are listed in this directory? What are their names?

(b) How many blocks does the file `makefs` occupy on the disk?

Q3 Given the root directory information from the previous question and the FAT table shown below:

```
0000200: 0000 0001 0000 0001 0000 0001 0000 0001 ................
0000210: 0000 0001 0000 0001 0000 0001 0000 0001 ................
0000220: 0000 0001 0000 0001 0000 0001 0000 0001 ................
0000230: 0000 0001 0000 0001 0000 0001 0000 0001 ................
0000240: 0000 0001 0000 0001 0000 0001 0000 0001 ................
0000250: 0000 0001 0000 0001 0000 0001 0000 0001 ................
0000260: 0000 0001 0000 0001 0000 0001 0000 0001 ................
0000270: 0000 0001 0000 0001 0000 0001 0000 0001 ................
0000280: 0000 0001 0000 0001 0000 0001 0000 0001 ................
0000290: 0000 0001 0000 0001 0000 0001 0000 0001 ................
00002a0: 0000 0001 0000 002a 0000 002b 0000 002c .......*...+...,
00002b0: 0000 002d 0000 002e 0000 002f 0000 0030 ...-......./...0
00002c0: ffff ffff 0000 0032 0000 0033 0000 0034 .......2...3...4
00002d0: 0000 0035 ffff ffff 0000 0037 0000 0038 ...5.......7...8
00002e0: 0000 0039 0000 003a 0000 003b 0000 003c ...9...:...;...<
00002f0: 0000 003d 0000 003e 0000 003f ffff ffff ...=...>...?....
```

(a) Which blocks does the file `foo.txt` occupy on the disk?

(b) Which blocks does the file `foo2.txt` occupy on the disk?