Simple File System (SFS)

The objective of this project is to understand how file systems are implemented. For this, we shall use a custom-made file system called SFS[1] and provide commands to access the stored files/directories.

Simple File System (SFS)

Our Simple File System (SFS) will be emulated on a disk file called "*sfs.disk*". You will be interacting with the disk file using the readSFS () and writeSFS () functions provided. Both of these functions operate on blocks of data (discussed later). The definitions are as follows.

int readSFS (int block_number, char buffer[1024])

-reads contents in the block given by *block_number* (zero based) from the disk file and stores it into *buffer*. The buffer passed in must be at least 1024 bytes long. The function returns 1 if the read is successful: otherwise zero (can happen if you specify an invalid block number).

int writeSFS (int block_number, char buffer[1024])

-writes the contents in buffer to the block indentified by block_number (zero based) in the disk file. The buffer passed in must be at least 1024 bytes long. The function returns 1 if the write is successful; otherwise zero (can happen if you specify an invalid block number).

In addition, a mountSFS ( ) function is provided that reads SFS metadata (number of blocks, number of inode entries, block bitmap, and inode table) into data structures in memory. See the given sfs.h file to get an idea on these structures. Try to make use of the structures. **Make sure** you call this function before starting to process the commands.

Structure of the disk file sfs.disk

The disk file is an emulation of a real disk drive. The file given to you has an exact size of 100KB. Since we are going to treat it as our disk drive, the file"s size should never increase or decrease. You will not have to worry about that since you shall always be using readSFS and writeSFS to access this file.

A block in this disk will be 1024 bytes. You will always be reading and writing in terms of char entries; so a block is essentially 1024 ASCII characters. Lets see how SFS is structured on this disk.

Block 0 (superblock) : BLBINB000000000000000000……

BLB = 3-digit number of entries in the block bitmap (coming up)

INB = 3-digit number of entries in the inode bitmap (coming up)

The remaining 1018 characters will be „0"

Block 1 (block bitmap) : A string of 0s and 1s

A value of 0 at the ith index indicates that block number i (zero based) is available; otherwise valid data exists in that block. Typically, only the first BLB number of entries in this block will be useful to you; ignore values at the remaining entries.

Block 2 (inode bitmap) : A string of 0s and 1s

A value of 0 at the ith index indicates that inode entry i (zero based) in the inode table (coming up) is available; otherwise valid data exists in that entry. Typically, only the first INB number of entries in this block will be useful to you; ignore values at the remaining entries.

Block 3 (inode table) : A sequence of inode entries (128 of them to be precise)

An inode entry is of the form :

TTXXYYZZ where

- TT = FI or DI
- XX, YY,ZZ= numbers between 04 and 99 (inclusive), or 00

FI means the entry is for a file. DI means the entry is for a directory.
XX, YY and ZZ are indices of the blocks that store the file/ directory contents. A value of 00 means the index is not used. So, an inode corresponding to an empty file/directory will have „00" for all three.

Since a file can at most take 3 blocks, its length in SFS can be at most 3*1024 = 3KB. Note that we are using 8 ASCII characters per inode entry; so the number of inode entries in the inode table is 1024/8 = 128 (=INB). The first entry (entry 0) is **always** for the root directory.

Block 4-99 : data

What will a block hold if the inode entry is for a directory?

When the first two characters of an inode entry is DI, it means blocks XX, YY and ZZ hold information about a directory. Such a block will contain 4 *directory entries* of the following structures :

F<name>MMM where
-F is either"1" (entry in use) or „0" (entry not in use)
-<name> is a 252 bytes (characters) long name given by the user to this directory entry
- MMM is the index of the inode entry in the inode table where information on this file (or directory) is available.

Each directory entry is therefore 1+252+3 =256 bytes. So, a block can hold at most 1024/256= 4 such entries. And since an inode entry can point to at most 3 blocks (XX, YY and ZZ), we can have at most 4*3=12 such entries per directory. This just means that SFS supports a maximum of 12 files (or directories) inside a directory!

What you are already given

Download the file *sfs.h* and he sample disk file *sfs.disk*. Go through the implementation in the .h file and try to reuse portions of the code. The code contains:

- **readSFS ( ), writeSFS ( )** and **mountSFS ( )**
- an implementation for a **1s** command
  -Lists the contents of current directory
- an implementation for a **cd <dir>** command
  -Changes the current directory (you cannot go up or more than one level down in the directory structure using this command)
- an implementation for a **md <dir>** command
  -Creates a directory called <dir> in the current directory (no nested paths are supported)
- an implementation for a **rd** command
  -Takes you to the highest level in the directory structure (current directory = root directory)
- an implementation for a stats command
  -Prints number of free disk blocks and free inode entries

- global variables :
  - **-int CD_INODE_ENTRY** = index of the inode entry in the inode table corresponding to the current directory
  - -**char*current_working_directory** = string with name of the current directory
  - -**int free_disk_blocks =** number of unused disk blocks
  - -**int free_inode_entries =** number of unused entries in the inode table
- global data structures to hold SFS metadata and some helper functions

What you need to do

Add a few more commands! You will accept the commands (those provided and those you would implement) from the standard input and process it until the command ***exit*** is entered.

TASK 1: implement a display file command

syntax:display <fname>

The command first checks to see if a file named *fname* exits in the current directory (of the disk File). If so, you should read the file from the disk file (using readSFS) and display its contents to standard output; otherwise, display an error message (do not terminate the program). *fname* will have no nested directories… so do not worry about something like /home/test/foo.txt.

TASK 2: implement a create file command

syntax: create <fname>

The command first checks to see (i) if file or directory with the name *fname* already exists in the current directory, or (ii) there is no available space for a new file (because the upper limit of 12 has been hit). If you, display and error message (do not terminate your program); otherwise, it reads some text from the standard input until the user hits ESC (ASCII code 27) or more than 3072 characters (3 KB) has been entered. At this point you must make sure you can successfully store the **entire** user input into the disk file (or else display an error). If you can, then a file called *fname* is created in the current directory (of the disk file) and the user input is stored as the contents of this file. You will be using and updating the block bitmap, inode bitmap and the inode table in this process, in addition to the directory entry. Remember to use readSFS and writeSFS!! Once again, *fname* will not have nested directories.

HINT : You can use almost all of the md command implementation for this!!

TASK 3: implement a remove file/directory command

syntax: rm <name>

The command first checks to see if name is a valid file or directory in the current directory. If not display an error message; otherwise remove the file/directory. This will also require making the disk blocks corresponding to the file/directory available (by updating the block bitmap) and making the corresponding inode entries in the inode table available (by updating the inode bitmap). Note that if the entry tor remove is a directory, then you should first recursively remove all content in that directory. Also, if the removed file /directory is the last directory entry in a block (there could be a maximum of 4), then you should return that block to the system.

NOTE: Your updates should become visible in the disk file as soon you finish processing a command. In other words, DO NOT wait until the exit command to write to the disk file. We may open two instances of your program and type the commands in any one of them!!

Error checking

Do not worry about checking errors other than those asked for in the create, display and rm commands.

Directory structure in the given disk file

```
/---
   l—text
   l      l     google
   l      l     sfs
   l  empty1
   l  empty2
   l--mickey
```

Example

```
prompt> ./a.out SFS::/
# 1s
text     empty1                    empty2                    mickey
1file and 3 directories.
SFS:: /# stats
88 blocks free.
121 inode entries free.
SFS:: /# md dump
SFS:: /# stats
87 blocks free
120 inode entries free.
SFS:: /# cd dump
SFS:: dump# create hello
(Max 3072 characters : hit ESC-ENTER to end )
Hello World!^[

12 bytes saved.
SFS:: dump# 1s
hello
1 file and 0 directory.
SFS:: dump# display hello
Hello World!
SFS:: dump# rd
SFS:: /# stats
85 blocks free.
119 inode entries free.
SFS:: /# rm dump
SFS::/# stats
88 blocks free.
121 inode entries free.
SFS::/# exit
prompt>
```