CSCI 4061
Assignment 3: File System
Due Date: 03/23/2017

Purpose of the assignment

The goal of the project is to strengthen the understanding of **File System** through a simplified implementation of the Unix file system.

Outline

With the confidence you have gained throughout your previous two projects, and the knowledge gained from your recent lecture notes and individual study, you are planning to take it to another level by saving the beautiful images you took during the winter holidays inside your own file system. However, after some more studies on the UNIX file system, you have kind of backed up from your initial ambitious plan and rather concentrating on making a miniature version of the file system.

To help you out, we came up with this assignment; we want you to code a simple, ***in-memory file system*** and similar to the previous assignments, you will work mainly with the image files. You have already converted all the images to jpeg files, and stored them in some directory, probably along with some other files. For this assignment, you will write a program that simulates an in-memory mini file system and some utility codes to help test the file system. You will be provided with a sample input directory, and some initial API's to follow with.

From the input directory, you will copy (***write*** to the file system) **all of the files** to your file system, provide a summary of the filesystem in a text file, and copy (**read** from the file system) only the ***images (jpeg)*** to an output directory. Then convert the images to thumbnails inside the same directory and prepare an html file showing all of the thumbnail images.

File System Structure

1. The file system will be in-memory and therefore, will exist only during the runtime of the code. Also, the filesystem will **only store files**, but not any directory.
2. The file system comprises of four components: Disk Blocks, Inode List, Directory Structure and Superblock. Each file inside the file system has a filename, inode number, and its corresponding inode list to point to the disk blocks where the content of the file is written. Have a graphical overview from ***Figure 1*****.**
3. **Disk blocks** are the in memory storage for data files. Total storage is divided into a fixed number of blocks. The number of blocks available are 8192 and each block is of 512 bytes.
4. **Inode List** stores, for each inode, the location of starting and ending blocks from the disk blocks, and some other information (see File Systems API). The number of inode is limited to 128.
5. **Superblock** holds the position of the next free inode and the next free disk block which

will help your to track the next place to write inside the Inode list, and the next disk block to use for storage.

6. **Directory structure** maps the file names to their corresponding inode number. Maximum number of files that this directory structure can hold is **128** (as you have already guessed).
7. For simplicity, every block is assigned to a file in its entirety. For example, if you have a file of size 510 bytes, you will assign 1 block (of size 512 bytes) to it, but If it exceeds 512 bytes (514 bytes for example), you have to assign 2 blocks (1024 bytes).

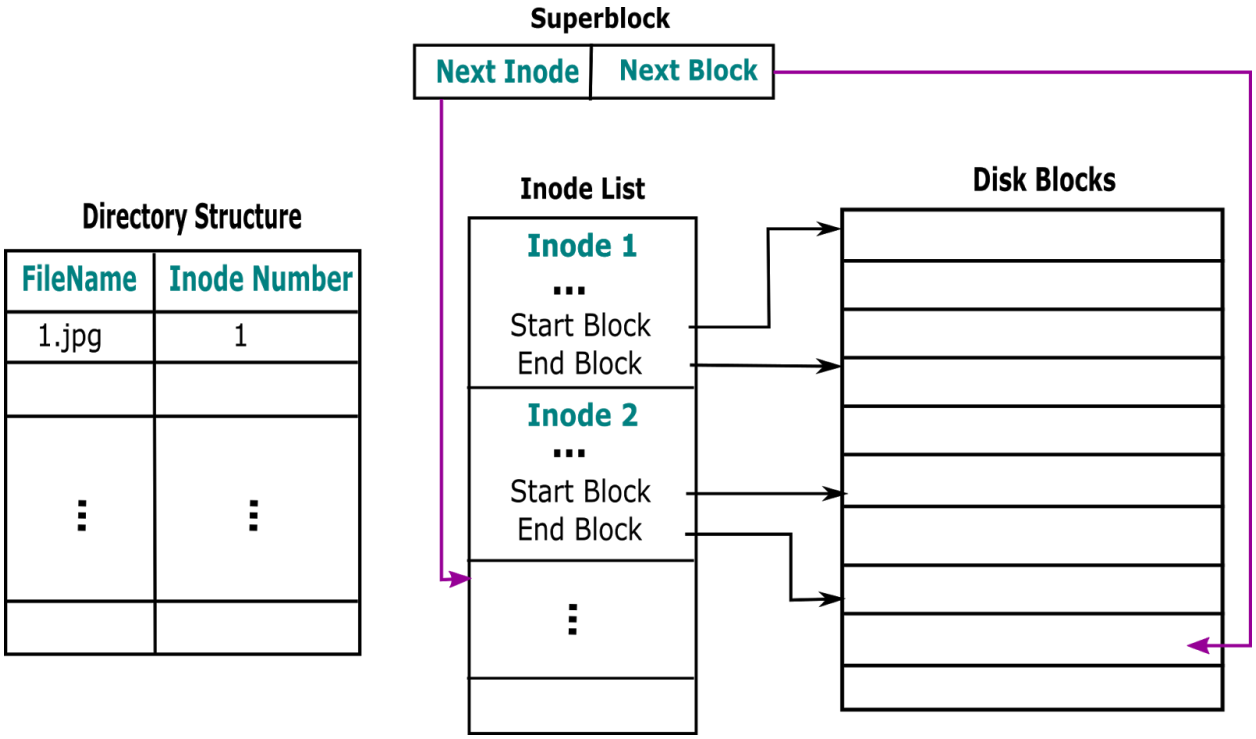

Figure 1**:* File System Structure with its **four** components: Directory Structure (left), Inode List (middle), Disk Blocks (right) and Superblock (top). All of the files inside the filesystem will have entries in **Directory Structure** (where filename and inode number of the file will be stored); detailed information of the file will be saved inside the **Inode List** along with the pointers to the real data blocks. **Disk Blocks** (512 bytes*** each) will store real data for all of the files; the maximum number of disk block is **8192**. The track of next free entry inside Inode List and next free block will be kept through the **Superblock**. The file system can store at most 128 files, which also gives the maximum number for Inodes also.

Programming Guidelines

You are provided with an **input directory**, where all of the image (jpeg) files (maybe along with some other files) reside. There may be directories inside the input directory, and you have to copy every files from there (unless there are duplicate files, in which case, you just need to copy

one of those). To help with the coding, a **File System API** (mini_filesystem.h) is provided which will consist of file system structures and API for necessary file systems functions. To implement these APIs, you will also need to write some core file systems functions (**File System Calls**), which are only accessible from File System API's and can't be accessed from any outside functions.

File system API

We have provided the file system API (mini_filesystem.h) which has declarations for the four file system components and the File system interface (The actual implementation might be done in mini_filesystem.c). There are also declarations for the log_filename (char *) and Count (int) in this file, so that these are accessible both to the file system implementation and your test code (discussed later in the assignment).

The four Components

1. **Superblock**: a structure; next Inode and next Block should be integers.
2. **Directory Structure**: an array of structures, where each structure holds:
    a. **Filename**: assume the maximum length of filename as **20** (**including the extension**).
    b. **Inode Number**: array index of the corresponding inode in the Inode_List.
3. **Inode List:** an array of structures, where each of them includes:
    a. **Inode number:** Array index of this inode in the inode list
    b. **User id:** This needs to be same as the user id of the original file
    c. **Group id:** This needs to be same as the group id of the original file
    d. **Size of the file in bytes:** Indicates the current size of the file. Updated for every write command, set to 0 by create command
    e. **Start block:** Array index of the first disk block that holds this file's data
    f. **End block:** Array index of the last disk block that holds this file's data
    g. **Flag:** Set it to 1, when the file is open, and 0 when the file is closed.
4. **Disk Blocks:** an array (of size 8192) of character pointers. The actual blocks (of size 512 bytes) can be allocated and assigned to the entries in this array per necessity.

File system Interface

To expose the file system API to your test code, you should have following functions declared in this file (We have already included these into the mini_filesystem.h file):

```
int Initialize_Filesystem (char* log_filen ame);
int Create_File (char* filename);
int Open_File (char* filename);
int Read_File (int inode_number, int offse t, int count, char*
to_read);
int Write_File (int inode_number, int offset, char* to_write);
int Close_File (int inode_number);
```

File system Implementation

The file system implementation (**mini_filesystem.c**) should provide the following two sets of functions:

File system Interface: These functions are declared as part of your API and can be called from your main program. **However, these functions cannot access the file system structures directly** (except for **Initialize_Filesystem**, as you need to initialize the file system structures inside this function). The functions **must use the lower level file system calls described later in this assignment** to change anything in the file system**.**

1. **Initialize_Filesystem:** Set the log file name as the filename provided as input. Set count to 0. Initialize (or allocate) anything else needed for your file system to work, for example, initial values for superblock. Return Success or Failure appropriately.

2. **Create_File:** Check whether the file you are going to create already exists in the directory structure. If yes, return with an appropriate error. If not, get the next free inode from the super block and create an entry for the file in the Inode_List. Then, using the returned inode number and filename, add the entry to the directory structure. Also, update the superblock since the next free inode index needs to be incremented. Then return appropriately.

3. **Open_File:** Search the directory for the provided file name. If found, set the inode flag for it to 1 and return the inode number. Otherwise, return appropriately.

4. **Read_File:** For the given inode number, check whether the provided offset and number of bytes to be read is correct by comparing it with the size of the file. If correct, read the provided number of bytes and store them in the provided character array; return the number of bytes read. Otherwise, return an appropriate error. **Hint:** You may need to make multiple calls to **block_read** (See File System Calls) to implement this.

5. **Write_File:** For the given inode number, first check if the provided offset is correct by comparing it with the size of the file (**Note:** a file is contiguous in this filesystem, you cannot write at any offset). If correct, write the provided string to the file blocks, update the **inode** (since this changes the **file size**, **last block** etc.) and **superblock** (as the **next free disk block** will change) with the right information and return the number of bytes written. If incorrect, return appropriately. **Hint**: You need to make multiple calls to **block_write** to implement this. You may do this in a loop:
    a. Fetch the Superblock to get next free disk block number
    b. Write to this block by calling Block Write
    c. Update Inode
    d. Update the Superblock

6. **Close_File:** For the given inode number, set the inode flag to 0 if it is 1 and return

appropriately.

File System Calls

This is a set of lower level functions that are not exposed to the user (or your test code) but these are the ones which can actually access and modify the file system structures. Use these to implement your file system API.

`int` **`Search_Directory`** `(char*` **`filename`**`)`

- Search through the directory structure for the given filename, return Inode number of the file, if the file is found and error (-1) if it is not.

`int` **`Add_to_Directory`** `(char*` **`filename`**`, int` **`inode_number`**`)`

- Add an entry to the directory structure with the **filename** and **inode_number** provided as input. Return the **status** indicating whether it was able to add the entry to the directory successfully or not.

`Inode` **`Inode_Read`** `(int` **`inode_number`**`)`

- For the given **inode_number**, if the file exists, return the **Inode structure** or **-1** if the file doesn't exist.

`int` **`Inode_Write`** `(int` **`inode_number`**`, Inode` **`input_inode`**`)`

- Copy the contents of Inode structure **input_inode** to the Inode present at the **inode_number**.

`int` **`Block_Read`** `(int block_number, int num_bytes , char* to_read )`

- Read the given **num_bytes** from the **block_number** and write it to the provided character String **to_read**; return the number of bytes read.

`int` **`Block_Write`** `(int block_number, int num_bytes, char* to_write)`

- Given the **block_number**, write the contents of the string **to_write** to the block and return the number of bytes written.

`Super_block` **`Superblock_Read`** `()`

- Return the superblock structure.

`int` **`Superblock_Write`** `(Super_block input_superblock)`

- Copy the contents of **input_superblock** to the superblock structure.

Testing the File System

Now that the file system implementation is done, the time for testing has come at last. Before writing your test code, we want you to make **a library file for the API and system calls** (See at the end of the document for details of library creation). From the test program, you will access

the necessary File System API through the library; to assist you on writing your test program, we are providing a dummy of the source code (test.c), and four interfaces to fill in:

`void` **`write_into_filesystem`**`(char* input_directory, char *log_filename)`

- This function will read all of the files inside the `input_directory` (which might be nested). Also it will provide the `log_filename`; check if the file exists, and if it does, remove and create a new one. Your program should first initialize the file system by making a call to Initialize_Filesystem by passing `log_filename` to it.

`void` **`make_filesystem_summary`**`(char* filename)`

- This function will make a summary of the filesystem and write into a file named `filename`. The format of the output file is flexible but it should include: **Filename**, **extension**, **Inode No**, and **Size**.

`void` **`read_images_from_filesystem_and_write_to_output_directory`**`(char* output_directory)`

- This function will read all the image (jpg) files from the filesystem and write into the `output_directory`.

`void` **`generate_html_file`**`(char* filename)`

- This function will convert the image files inside output directory to thumbnail images, write into the same directory and prepare a html file using the thumbnails.

After the test code is written, compile it using the library and make an executable. The executable can be used like: `<executable-name> <input_dir> <output_dir> <log_filename>` (for example, `./test input_dir output_dir log.txt`).

Generic points

1. For each file created, there should be a corresponding entry in the directory structure and an inode in the inode list.
2. The files are to be laid out contiguously on the disk, one after the other and the inode should hold the start and end block number indicating the first file block and last file block.
3. You should log an entry to the log file for every access made to any of the file structure. See the **logging** section below for more details.
4. Your file system should keep track of the number of accesses made to it. At the end of your test program, you should print this Count.
5. You should increment the count variable for every access made to any of the file structure. This can be easily achieved by incrementing the count every time a call is made to the low level function.
6. Status and return values can follow the following pattern:
    a. -1 or NULL => Error
    b. 0 or the value requested => Success

Test Data: A compressed tar file is provided with this Assignment. Note that your input directory can be nested.

Logging: For every access done to the mini file system, be it superblock, directory structure, inode and disk block (an entry for every disk block access, and not every file access), add an entry to the log file. You can accomplish this by adding entries from the lower level file system calls implemented by you. Each entry in the log file should have the following format:
<Timestamp> <File structure Accessed> <Read/Write>
Time Stamp may be in any human readable format but has to include microseconds (you can use **hr:min:sec:micro-secs** if you can't decide).

Output Directory and HTML file

After you have copied the images from your mini file system to the output directory, follow the steps below to create the HTML file:

1. Create thumbnail image for each of the image in this output directory. You can save them in the output directory itself. The name of the thumbnail image should be <current_filename>_thumb.jpg Note that thumbnail images must all be 200 pixels in the largest dimension (either width or height)
2. Using the contents of output directory (Images and thumbnails), create an HTML file with the name **filesystem_content.html** (below is an example file)

```
<html>
      <head>
            <title>Image 01</title>
      </head>
      <body>
            <a href="images/sunset.jpg">
            <img src="images/thumbs/sunset_thumb.jpg"/></a>
      </body>
</html>
```

Experiment

Once you have the file system setup and all the files written to the mini file system, you will be performing read operation on the files from your test program. To get the sense of approximately how much impact a defragmentation operation can create, you will use your program to count the number of accesses made to the mini file system. This can be easily accounted for using the **"Count"** Variable and by incrementing it once in every low level function. This count accounts for **both read and write** operation. Half it (to account approximately for read operation) and then take average over total number of files. This should give you the **average number of access made per file read**. With the value calculated for

average number of access made to the file system per file read, calculate the best case and average case time required for reading a file.

Best case – When the file system is appropriately and contiguously laid.
Average case – When the file system is fragmented and the blocks are mapped randomly to the disk sectors.

Use the following disk parameters for this calculation:

- Rotation Rate – 15,000 RPM
- Average Seek time – 4 ms
- Average number of sectors per track – 1000
- Sector size = Block size = 512 bytes.

Assume that the time to position the head over the first block is Average Seek Time + Average Rotation time.

Error Handling

- Errors must be handled gracefully, with informative error messages on standard error.
- You cannot count on your user always giving you the right number of arguments. If the wrong number of arguments is given, then your program should print a usage message and exit with a status of 1.
- You must also check for errors on the system calls you use, and use the **perror**() function to report them. Function calls will be needed for getting the file related data while populating the inodes.
- You may add your own error reporting if you feel it was necessary.

Platform

You may work on the platform of your choice: Linux, Solaris, MacOSX or Windows. However, you need to ensure that your script works correctly on one of the CSELabs UNIX machines, where we will grade your assignments.

Teamwork

You may do this assignment individually or with one partner.

Deliverables

See the course syllabus for general requirements and assignment page for submission guideline. The specific requirements of this assignment are as following:

1. README.txt should include
    a. Personal information for the group
    b. CSELab machine you tested your code on
    c. Syntax and usage pointers for your program

    d. Any special test cases or anomalies handled/not handled by your program.
    e. Experiment results
2. Source code and library files. All *.h, *.c files required to execute your program. Also add the static library file and a Makefile to create an executable using the library and test program.
3. Your commentary as described in the syllabus. This should explain, briefly, what your code does, how it works, how you use it, and how to interpret its output. It should be no longer than TWO pages in length.

Grading

In this assignment, 90% is allocated to the code (of which, 20% on coding style, 20% on output and log file and 50% on correctness), and 10% is allocated to the experiment. The general grading criteria are given in the course syllabus. For this exercise the coding style points and formatting points will be based on readability. Use meaningful names if you create variables, and use consistent indentation to display any control structure in your code. Follow the following coding styles:

- Informative description at beginning of the program (5pts)
- Informative comments within the program (5pts)
- Use meaningful names if you create variables (5pts)
- Use consistent indentation to display any control structure in your script. (5pts)

Resources and Hints

1. For understanding how a UNIX file system works – Chapter 5 from the book “The design of the UNIX operating system” by Maurice J Bach. This Assignment requires much simplified version of the actual implementation of the UNIX file system, and you would not require the in depth understanding the system calls this chapter covers, but it’s a good read if you want to understand how the actual file system works and how it uses some sophisticated data structures and algorithm to be efficient.
2. **For experimentation:** You can refer to example problems provided in the textbook (Computer Systems: A Programmer's Perspective) of CSCI 2021 (Check out the section on **Disk operation** of **Chapter 6**: The memory hierarchy).
3. Note that, you are not performing reads and writes to a character file while dealing with input and output directory but to an image file but you will be writing to a character array [512 bytes block]. So, you should be careful with your reads and writes. Specially while dealing with EOF, which is better indicated as integer and not character.
4. Take this as a design problem and not just as a programming assignment. You should think about issues that your program can run into, and design your functions accordingly! Few design considerations:
    a. Since the size of all the file system structures are of fixed size, you can use separate arrays for each of them instead of using a linked list. This will make your program time efficient. But I would suggest on using pointer arrays, and allocating space for each structure as you need, for being space efficient.
    b. For consistency of the file system (superblock being the major concern),

synchronization methods are used. Since we have not covered that yet as part of the course, you should make sure that you have only one file open at a time. Finish with all the operation with that file, close it and then open another file.

Create a static library from the source code

For this project, we would also learn how to make a library file (for example, if you want people to use your work, but are skeptical to reveal the source code). After you are finished writing your APIs for file system, archive those into a library file and use the library to test your program. Below is the instruction of making a static library (*.a files in UNIX).

Assume that we have two different files test1.c and test2.c, and we want to create a library from these source files. To generate the library, you can use:

- `cc -c test1.c test2.c`
- `ar -cvq libctest.a test1.o test2.o`

Now, to link the library to use with a test code (say main.c):

- `cc -o <executable_name> main.c libctest.a`

Additionally, to look inside the library:

- `ar -t libctest.a` -> List files in library
- `nm -C libctest.a` -> Show the function names used inside the library

We have tested sample codes in KH4-250-33 to check the necessary tools to build a library and executable using the same library works correctly.

Answers of Frequently Asked Questions

- Input directory can have directories inside them, which you should parse through and gather the list of all file types. **Caution:** There can be two files of the same name in different directories, your program should handle situations like these correctly.
- You can assume that filename will not be larger than **20** characters; for example, the length of the filename, **abc.jpg**, is **7**.
- You can assume that input files exist and are readable.
- You may assume that none of the input file names will have any spaces in the name.
- You can assume that all file names will be in lowercase format (.jpg for example)
- You may assume total file count cannot exceed 128 or the total size would not exceed the size of the mini file system. But it is advisable to have error checks around situations like these.
- There are no soft / hard links in the directory tree. There are only files.
- Any message (i.e. for debugging) can be redirected to the terminal.
- You can write the **log file, summary file,** and **html_file** in the working directory or output directory; just mention it in the readme file.