Unix File System

DUE:

Due on Saturday 03/28/2015 by 11:59 PM (Online, no hard copy)

PURPOSE:

The purpose of this assignment is to write a C program that simulates a simplified version of UNIX File System.

SCENARIO:

Continuing with the theme of previous assignments, over time, you have accumulated a lot of image files, and these are spread across your file system. Creation and deletion of files has led to fragmentation and your file system has become very slow. Fed up with this, you want to reorganize your files and this time in your own mini file system.

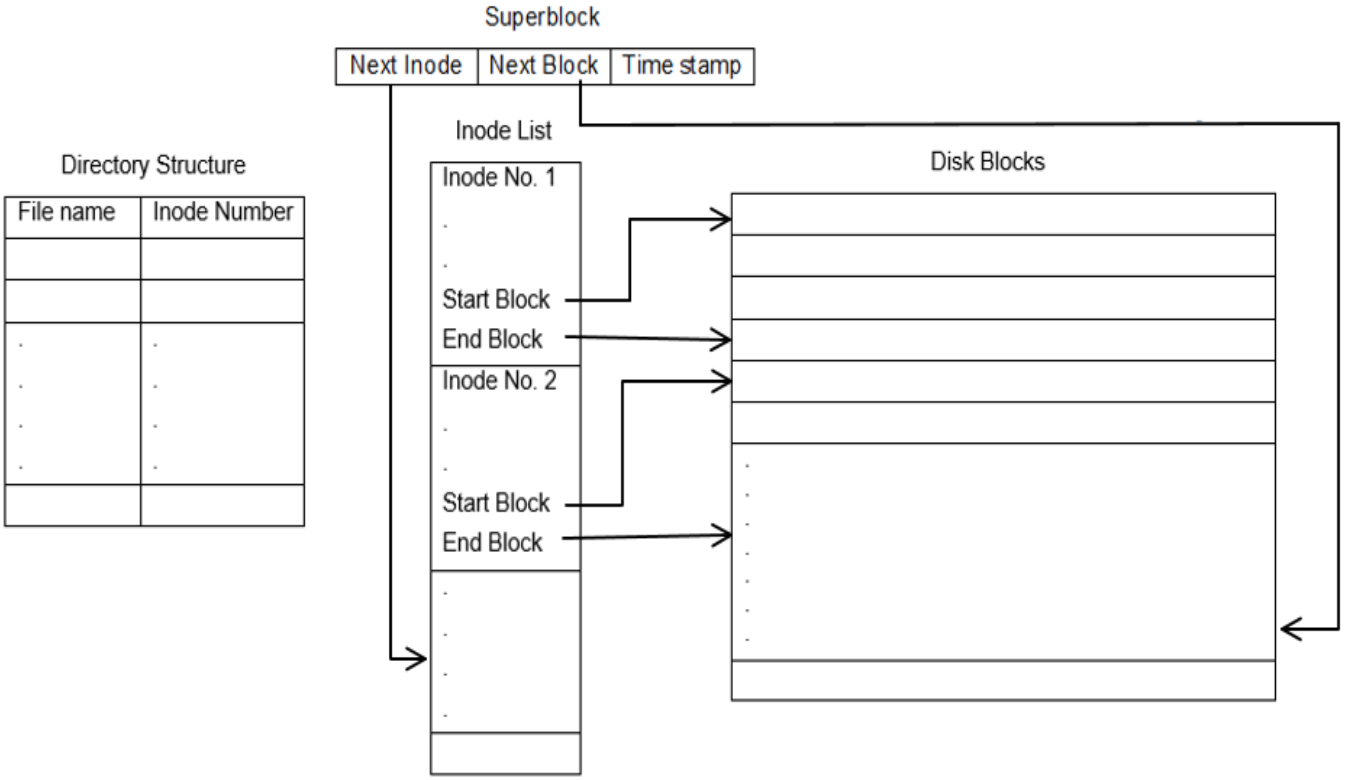
For this assignment, you will write a program that simulates this mini file system and another program that will test this file system.

You are provided with a sample input directory with this assignment. Your test program should copy all the image files from this input directory to your mini file system – this will test the write feature of your file system.

Next, to verify the correctness of your file system, as well as to test the read feature, your test program should read the files one after the other from this new file system using the file name and copy them to an output directory. Further, you need to create an HTML file using this output directory.

MINI FILE SYSTEM STRUCTURE:

1) Since we are simulating, this will be an in-memory file system.
2) There are 4 components of your file system, Superblock, Directory structure, Inode List and the Disk Blocks.
3) Your Superblock will hold the position of a next free inode and the next free disk block.
4) Your Directory structure will map the filenames to their corresponding Inode number. Maximum number of files that this directory structure can hold should be limited to 128. Note that this is a one level directory structure and will hold only files and not directories.
5) Your Inode List will have the list of inodes and number of inodes is also limited to 128. Each inode will hold a set of information for each file as listed in the programming guidelines.
6) Your disk blocks is where you will store the actual file data. The total storage is divided into blocks. Total number of blocks available are 8192 and each block is of size 512 bytes. You need to allocate memory accordingly.
7) 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. If it is of size 514 bytes, you will assign 2 blocks (1024bytes).



PROGRAMMING GUIDELINES:

Components:

1) **File system API : mini_filesystem.h**
2) **File System Implementation: mini_filesystem.c**
3) **Test Program: test.c**

File system API:

Your file system API (mini_filesystem.h) should have declarations for the four file system components and the File system interface. The actual implementation should be done in mini_filesystem.c file. You should also declare the log_filename (char *) and Count (int) in this file, so that it is accessible both to the file system implementation and your test program (We will discuss what these are needed for later in the assignment). We have provided mini_filesystem.h with this assignment for your reference.

The four Components:

1) Superblock: Declare it as a structure. Next Inode and Next Block should be integers.

2) Directory Structure: Declare it as an array if structures. Each structure will hold following details for each file.
   1) Filename < Character string. You can assume the maximum length of filename as 15 >
   2) Inode Number <Integer: Array index of the corresponding inode in the Inode_List >

3) Inode List: Declare it as an array of structures. The structure should hold following details for each file:
   1) Inode number < Integer: Array index of this inode in the inode list>
   2) User id < Integer: This needs to be same as the user id of the original file >
   3) Group id <Integer: This needs to be same as the group id of the original file>
   4) Size of the file in bytes <Integer: Indicates the current size of the file. Updated for every write command, set to 0 by create command>
   5) Start block < Integer: Array index of the first disk block that holds this file's data>
   6) End block < Integer: Array index of the last disk block that holds this file's data>
   7) Flag <Integer: Set to 1, when the file is open, and 0 when the file is closed>

4) Disk Blocks: Declare it as 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 later as need arrives.

File system Interface: To expose the file system API to your test program, you should have following functions declared in this file:

1) Initialize_Filesystem
   a. Input: (char *) log_filename
   b. Output: (int) Status
2) Create_File:
   a. Input: (char *) Filename
   b. Output: (int) Inode number
3) Open_File:
   a. Input: (char *) Filename
   b. Output: (int) Inode number
4) Read_File
   a. Input: (int) Inode number, (int) offset, (int) number of bytes, (char *) String to read file in.
   b. Output: (int) Number of bytes read.
5) Write_File
   a. Input: (int) Inode number, (int) offset, (char *)String to write
   b. Output: (int) Number of bytes written
6) Close_file
   a. Input: (int) Inode number
   b. Output: (int) Status

File system Implementation:

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

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

1) Initialize_Filesystem
   Implementation: Set the log file name as the filename provided as input. Set count to 0. Initialize anything else if you think is needed for your file system to work, for example, set initial values for superblock, allocate/initialize anything else that you may need etc. Return Success or Failure appropriately.
2) Create_File:
   Implementation: This function should first check whether the file with the provided file name already exists in the directory structure. If yes, return with appropriate error. If not, get the next free inode number from the super block and using that create an entry for the file in the Inode_List. Then, using the inode number returned and filename you have, 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:
   Implementation: Search the directory for the provided file name. If found, set the inode flag for it to 1 and return the inode number. Else, return appropriately.
4) Read_File
   Implementation: For the given inode number, first check if 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. If incorrect, return appropriate error. Hint: You need to make multiple calls to block_read to implement this.
5) Write_File
   Implementation: 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 can do this in a loop:
   a) Fetch Superblock to get next free disk block number
   b) Write to this block by calling Block Write
   c) Update Inode
   d) Update Superblock
6) Close_file
   Implementation: For the given inode number, set the inode flag to 0 if its set to 1 and return appropriately.

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

1) Search_Directory
   a. Input: (char*) Filename
   b. Output: (int) Inode Number
   c. Implementation: This should search through the directory structure for the given file name.

It returns Inode number of the file, if the file is found and error (-1) if it is not.

2) Add_to_Directory
    a. Input: (char *) Filename, (int) Inode Number
    b. Output: (int) Status
    c. Implementation: This should add an entry to the directory structure with the Filename and Inode number provided as input. It returns the status indicating if it was able to successfully add the entry to the directory or not.
3) Inode_Read
    a. Input: (int) Inode number
    b. Output: (Inode Structure) Inode
    c. Implementation: For the given inode number, if the file exists, this function should return the Inode structure and NULL if the file doesn't exist.
4) Inode_Write
    a. Input: (int) Inode number, (Inode Structure) Inode
    b. Output: (int) Status indicating successful write or not
    c. Implementation: This function should copy the contents of Inode structure provided as input to the Inode present at the Inode Number passed.
5) Block_Read
    a. Input: (int) Block number, (int) Number of bytes, (char *)String to read in
    b. Output: (int) Number of bytes read
    c. Implementation: This function should read the given number of bytes from the provided block number and write it to character string provided. It should then return the number of bytes read.
6) Block_Write
    a. Input: (int) Block number, (char *)String to write
    b. Output: (int) Number of bytes written
    c. Implementation: Given the block number, write the contents of the string provided to the block and return the number of bytes written.
7) Superblock_Read
    a. Input: Nothing
    b. Output: (Superblock structure) superblock
    c. Implementation: Return the superblock structure.
8) Superblock_Write
    a. Input: (Superblock structure) superblock
    b. Output: (int) Status whether superblock was successfully written
    c. Implementation: Copy the contents of the provided superblock structure to the superblock.

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 section on logging for more details.

4) 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.
5) Status and return values can follow the following pattern:
    a. -1 or NULL => Error
    b. 0 or the value requested => Success

General Flow of your Test Program:

1) Your program will take as input the name of the input_directory, an output_directory and a log_filename.
   a) Intput_directory – Check if it exists, if it doesn't, you should exit with appropriate error message.
   b) Output_directory – Check if it exists, if it does, use that, if it doesn't, create one.
   c) Log_filename – Check if it exists, if it does, remove and create a new one.

2) Your program should first initialize the file system by making a call to Initialize_Filesystem by passing log_filename to it.

3) Your program should then parse through the input_directory to identify all the image files of the format – JPG. Note that your input directory can be nested.

4) Your program should then write each of these files to your mini file system using the file system interface developed by you.

5) Once it's finished writing, your program should read each file from your mini file system (again, using the interface developed by you) by referring them by their filename and copy them to the output directory.

6) Your program should then prepare an HTML file with these images.

7) Your file system keeps track of the number of accesses made to it. At the end of your test program, you should print this Count.

Program Usage: Make sure your final executable is of the following format including the name of the executable for testing and grading purposes

Usage: test <input_dir> <output_dir> <log_filename>

TEST DATA:

A compressed tar file is provided with this Assignment. Note that your input directory can be nested. You can extract a compressed tar file with the command:

tar xzf <tarfile>

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 in the format described later in the assignment. You can accomplish this by adding entries from the lower level file

system calls implemented by you.

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
   Here's an example of filesystem_content.html:

```
<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 need to perform the following experiment.

For this experiment, you will use your program to count the no. 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. Once you have this, use it to calculate the required experiment results as explained later in the assignment.

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.

VALID ASSUMPTIONS:

- Input directory can have directories in them, you should parse through and gather the list of specified 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 file name will not be larger than 15 characters.
- 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 may assume total file count won't 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.

PLATFORM:

You may work initially on the platform of your choice; Linux, Solaris, and Mac OSX should all work. However, you must test under CSELabs UNIX Machines Linux box (machine list is here), and we will test all submissions using Linux on one of the boxes when we grade your code.

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:

You need to include the following files for this assignment:

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 files. All *.h, *.c files required to execute your program. Your main program should be part of the file test.c

3. Makefile for compilation. The make file should compile and link files to produce the executable that can be used in the format provided in the program usage requirement.

4. 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.

5. Your code will be tested by another program. To make sure that your information is read correctly, the beginning of test.c file must adhere to the format that is given below. If you are not working in a group you should specify only one name and id.

    ## Information
    CSci 4061 Spring 2015 Assignment 3
    Name1=FullName
    Name2=FullName
    StudentID1=ID for First name

StudentID2=ID for Second name
Commentary=description of the program, problems encountered while coding, etc.##

PROGRAM USAGE REQUIREMENT:

username@Directory/path:$ make

username@Directory/path:$ test [input_dir] [output_dir] [log_filename]

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.

EXPERIMENT RESULTS:

With the value calculated for average number of access made to the file system per file read, calculate the best case and worst 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.

CODING STYLE:

- 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)

LOG FILE FORMATTING:

Each entry in the log file should have the following format:

<Time Stamp> <File structure Accessed> <Read/Write>

NOTE: Test data used while grading will not be provided.

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 text book of CSCI 2021.

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 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:
   1) 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.

   2) 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.

Note: This is a design assignment, and not every instruction will be laid out for you. You can take your own decisions at places and we do not care about very minute details. As long as you follow the instructions specified in this assignment and implement the functionality well, you should be good with grades.