Lab 5: File Systems
CMSC 442

A file system organizes data stored on a disk or other storage device at two different levels: the internal or physical arrangement of the blocks of a file and the external or logical directory structure in which files are organized for the user's benefit. In this lab, you will implement a simple (but very inefficient) file system in C++. Your file system will have only one directory and will assume that each file can fit into a single 1024 byte block (in other words, no file will be more than 1024 bytes long).

While the data stored by most real file systems resides on a hard drive or other device, your file system will store its data in a single huge array of characters. The entire array will be saved to – and restored from – a regular (Linux) data file stored on the native file system of your computer so it will be persistent and can be copied from machine to machine.

Checking out the initial code

I have provided code for you in a git repository on torvalds. You can check it out into your home directory or other suitable location by doing a "cd" to that folder and then typing:

git clone /home/robert/OSLab5

The code I provide for you consists of a C++ class named "Filesystem" and a series of test files that check each of the operations you are required to implement. I also provide a convenient Makefile which you can use to build and test your project.

The Filesystem object contains an array of characters named *m_disk* that you will use as your virtual disk. The size of the disk is stored in an unsigned integer named *m_disk_size*. By default, the disk is 4Mb in size.

The Filesystem object contains function stubs for ten file system operations you will need to implement. The code for these should go in "operations.cpp". Prototypes for each function stub are defined in "filesys.h", **which you should read thoroughly before starting to code anything**. Make sure you understand the parameters each function takes and what it should return, then implement the function following the instructions given below in this handout. Some functions may require modifying the Filesystem class to add additional data structures or (private) variables.

You can copy information into the array just as you would in any C++ program. For instance, if you wanted to write the string "hello" starting at position 10 of the array, you could do something like:

```
m_disk[10] = 'h';
m_disk[11] = 'e';
m_disk[12] = 'l';
m_disk[13] = 'l';
m_disk[14] = 'o';
```

Of course, this is pretty inefficient. A better approach is to simply use a loop:

```
string str = "hello";
for (int i=10; i < 10 + str.length(); ++i) {
   m_disk[i]=str[i];
}
```

Because you will need to copy data into and out of the array a lot, I have provided you with some helper functions that will make this easier. The function read_string takes a position of the disk to read from, a string to read into, and a size. It reads "size" characters into the string. Similarly, the write_string function takes a

disk position, a string to write, and a size. It writes “size” characters from the input string onto the disk at the specified position.

Keep in mind that this data is stored “directly to the disk”. The helper functions don't have any concept of a file and so they will happily write data past the end of a file or into “free” space where no file has been allocated. Creating data structures to represent the files and directories on the disk is part of your goal for this assignment.

Because m_disk is an array of characters (essentially a string), storing other kinds of data, such as integers, can be tricky. The best solution to this problem is to use casting to trick C++ into thinking that an appropriate number of characters is really an integer. I have written two helper functions named read_int and write_int that implement this idea. You should read through them carefully to be sure you understand how they work. You should write similar helper functions of your own to read and write other data items, such as directory entries, to the file system.

Creating a disk image

The constructor of the Filesystem object takes a string parameter – the name of the Linux file to read into memory as your virtual disk – and an unsigned integer – the size of the file. If you do not specify a size, the program creates one that is 4Mb long. You can also use a pre-existing file. One way to make a file of a particular size is to use the dd command like this:

```
dd if=/dev/zero of=disk.img bs=1024 count=1024
```

This creates a file that contains 1024 blocks of 1024 bytes each in it (in other words, a -one- megabyte file). You can adjust the “count” parameter to obtain various sizes. The “of” parameter is the “output file name” – the name of the file to create.

The destructor in Filesystem saves the array into the file before destroying it. If you allocate your Filesystem objects statically in your test programs, this process should happen for you automatically. If you instead allocate them on the heap with “new”, you must be careful to “delete” the object before terminating the program so that the data will be saved to the disk. The provided test cases do this for you.

Group Task

The group work for this lab is divided into two sections of about five steps each. Each step is worth 5 points (for a total of 50 points). The remaining 50 points come from your individual task. The first section involves implementing a basic file system with one directory and single-block files. The second section involves implementing file operations such as read and write.

I. A Basic File System

In this section, you will create a file system in which all files share the same directory. We will assume that all files can fit into a single 1024 byte block. We will also observe the following limitations:

1. Filenames are case-sensitive, but will contain only letters, numbers, underscores, and the dot character.
2. Filenames will be a maximum of 15 characters long (but will take 16 bytes – one for the NUL terminator).
3. There will never be more than 4000 files in the file system.
4. Filenames are globally unique.

We will reserve space at the beginning of the file system for use as the “root directory” of the filesystem. The remaining bytes will be used to store the blocks of each file. The directory will consist of a series of “rows”, where each row corresponds to exactly one file, and will contain the filename, a four-byte integer index which

tells on which block the file starts, and a four-byte integer representing the size of the file.
In other words, each entry consists of 24 bytes: the filename (which can take up to 16 bytes), the index (4 bytes), and the size (4 more bytes).

Since each block of data takes 1024 bytes, the total space we need to store one file is 1024 + 24 or 1048 bytes. That means that the maximum number of files we can have is m_disk_size divided by 1048.

For a 4Mb file system, the number of possible files is 4002, so we would store the directory in bytes 0 through N - 1 = 96047 of the m_disk array and the first file block would start at position 96048.

A. Formatting the File System

We need some way of telling whether the values stored in each directory entry actually refer to real files or if they are simply garbage bytes. The easiest (though definitely not the 'best') way to do this is to "format" the file system by copying some sort of sentinel character over the entire directory record. Since filenames cannot contain asterisk characters, we can use those for our sentinel.

In the "format" function of operations.cpp add code that:

1. Calculates how big the directory record needs to be.
2. Copies asterisk characters into the over the entire directory structure of the m_disk array.

*Hint: You can caluate the space needed for the directory record using the formula (m_disk_size / 1048) * 24. (Each block takes 1024 bytes, since each directory entry takes 24 additional bytes, we need 1048 bytes for each entry). For example, a default image that is 4Mb would need (4194304 / 1048)*24 = 96048 bytes. Be careful about rounding.*

B. Creating a File

Creating a file really just means finding a free block and creating a directory entry that maps the filename to that block. You can divide this task naturally into three pieces:

1. Find a free space in the directory listing for your file.

2. Find a free block on the disk for your file. This is the trickiest part. We don't want to reuse a block that is being used by another file and we don't have a FAT table or inode table, so we have to use the directory to find a free block. The directory might not list the files in the same order that the blocks appear in the data region. That can happen if a file is deleted (delete can move the directory entries around).

I suggest the following approach:

1. If there are no files in the directory (this is the first file, or we have deleted all the files), we allocate the block that starts immediately after the directory.

2. If there are already files in the directory, use a loop to scan through the directory and calculate the "maximum" allocated block. Allocate the block which follows that "maximum" block. For now, you may assume that the block will "fit" inside the disk image.

3. Insert the filename, start position, and size into the directory listing.

Hint: Make a struct which represents a directory entry. Then cast the m_disk array to be an array of directory entries. You can then loop through those entries to check positions and/or insert your entry.

Your create function should return false if:

1. The directory is full and there is no more room for a file to be created.
2. The name of the new file matches the name of an existing file.
3. It is not possible to allocate a block for the file (for instance if the disk is full).

Otherwise, it should return true.

C. Listing the files in a directory

The "list" function should create a nicely formatted listing of all the files you have created and their sizes. It should return that listing as a string. It does not need to **(and should not)** include the position of the file since this is an implementation detail and not something that should be exposed to the user.

Each entry of the directory list should be on its own line. The filename should appear, left-justified, in a column 16 characters wide, padded with spaces. The size should follow, left-justified, in a column 8 characters wide, padded with spaces. You do not need to (and should not) add commas or units to the size – simply use the number of bytes.

Be sure not to include any deleted or not yet created files. (You may want to come back to this part after doing step D).

Hint: As for create, your best bet here is to cast the m_disk array to an array of directory entry objects and loop through them.

D. Deleting a File

To delete a file, we need to remove it from the directory. We can remove it by zeroing the filename entry out with * characters, but if we do that the entries that come after it could become unreachable (depending on how you implemented your list and create functions). A different solution is to swap the last entry of the directory with the one we are deleting and write the asterisks over the duplicate last entry. However, as long as your delete works properly (the file is removed from the directory listing and we can reuse its space for new files), without breaking any of the other functions, you can take any approach you want.

Write code in the "rm" function that takes a string, the filename, as a parameter and removes that file from the directory if it exists.

The rm function should return false if the file to be deleted does not exist. It should return true if it is able to successfully delete the file. You do not need to erase any data from the actual data blocks.

E. Renaming a File

The "rename" function should take two parameters, an old name and a new name.

If a file with the old filename exists and no other file is already using the new name, it should modify the directory listing to replace the old name with the new one. To do this, you will need to modify only the directory listing, not the data blocks of the file.

Hint: The easiest solution to this task is to loop through the directory multiple times (for each validity check) using the same casting trick as before.

F. Copying a File

To copy a file, we need to create a new directory entry and then copy the actual data from the first file's data block to the second file's data block. One way to implement this is to use the "create" function you wrote in step B and then copy the data. However, you do need to do error checking to be sure that:

1. The source file exists.
2. There is space available (both in the directory and the disk array) for the copy.

You should return false on error, true otherwise.

II. File Operations

All of the functions we have written so far directly affect the external layout and the way files are presented to the user. They don't directly allow you to access the data within a file. To actually store information in a file, we need to be able to read from the file and write to it.

We can do that more conveniently if we use "file handles" to keep some state information. We can manage the file handles using functions named "open" and "close". Open and close are really just convenience functions. They make it easier to read and write so that you don't have to remember a filename and file position every time you access a file.

A. Opening a File

The open function creates a "file handle" object that can be used in the read and write functions. To help you out a bit, I have created a FileHandle class for you in handle.h. It has no functions – it's really just a "struct" that keeps track of three things:

a. The position (in m_disk) of the start of the file
b. The size of the file.
c. An offset that keeps track of where we left off the last time we read from the file or wrote to it.

The open function should:

1. Create a new file handle object.

2. Set the file handle's size and position fields by looking the filename up in the directory and copying values out of the directory entry.

3. Initially, set the offset to 0. (The "file cursor" starts at the beginning of the file)

4. Insert the file handle into some sort of list or array so that it can be used by read, write, and close. You will need to create this list. I recommend using a C++ vector, which you can declare at the top of operations.cpp.

5. Return a file descriptor which uniquely identifies the file handle.

The key part of this step is to create and manage sort sort of array or list. We will call this the "open file list". It's probably best, actually, to store pointers to file handles rather than actual file handles – so that you can free the memory for the handle when you close the file. If you, instead, use an array, you will need an integer to keep track of the size of the list. You may assume that there are never more than 256 files open at a time.

B. Closing a File

Closing a file means freeing the memory used by its file descriptor. If you implemented open correctly, it should be trivial. You just need to delete the memory you allocated for the file handle.

You do not need to worry about making the file descriptor available for new files. None of my tests will ever open more than 256 files (except to test that your open function correctly bails out when that happens). Close should return false if the file descriptor isn't valid (does not correspond to a file handle that is open). Otherwise, it should return true.

C. Reading from a File

Now that you can create a file and open it up, you should be able to write code that reads a string from the file. The read function takes three parameters, the file descriptor to read from, a reference to the string which the data should be placed in (the "buffer" string), and the number of bytes to read.

Write code that:

1. Uses the file descriptor to obtain the file handle.
2. From the file handle, obtains the position of the block where the data is stored.
3. Adds the "offset" field to the position to obtain the location in the array where the data is located.
4. Reads "size" characters from that position of the m_disk array into the buffer string.

You should return false if the file descriptor is invalid or if reading would go past the end of the virtual disk. You should return true if the read succeeds.

D. Writing to a File

Writing to a file is almost identical to reading from one. You need to use the file descriptor to get the file handle, then use the position and offset from the file handle to locate the position at which to write. One difference is that when you write, the size of the file may change. If the last byte you write goes beyond the current size of the file, you will need to update the size in the directory entry. There are several ways to do this. One way is to add an integer to your file handle that keeps track of which row of the directory the corresponding file is located at.

The write function should return true if it succeeded and false if writing fails (for instance, if the disk is full or writing would overwrite some other file). If should also return false if the file descriptor is invalid.

Documentation (20 points)

Please create a file named README-LAB5.TXT which contains the following:

1. Your group name and the names of your group members
2. A description of the individual task each member contributed and how I can run/test it.

Individual task: Glitter (30 pts)

I would like each of you to pick a different way you can extend this project. This could consist either of adding new functions (with test cases) to the project or improving the efficiency of the existing functions. Here are some glitter ideas: subdirectories, multi-block files, giving files read-only permissions, giving files an "owner" string and adding "owner" and "world" permissions, creating a function that "defragments" the data blocks and the directory to reduce external fragmentation. Be sure to document your work.

Testing and Submitting

In addition to the code I provided for you, I have given you a folder named 'tests' that will allow you to check whether or not your functions work correctly. There is one file per function. Each file contains a set of unit tests that check both whether the operation can complete successfully and if the function returns the right values when an error occurs. The tests are incremental in the sense that each test relies on your passing the previous tests. For example, you must have format working before the test for “create” is meaningful.

The tests are not comprehensive, but they will at least give you some idea of whether your code works properly.

You can run the tests with “make test”. To submit, create a tarball and upload it to the course web site as usual.