Project

Due Dec 13 at 11:59pm **Points** 100 **Questions** 1
Available Oct 30 at 12am - Dec 13 at 11:59pm about 2 months **Time Limit** None
Allowed Attempts Unlimited

Instructions

Introduction

In the next several class sittings you will be working on implementing an in-memory file system simulator. You will then integrate it with **FUSE** (http://fuse.sourceforge.net/) to allow users to interact with your simulated file system using the Linux file system interface. There will be no distinction between interacting with the simulated file system and any other file system available to the users in the operating system. You will be able to explore the simulated file system, create and delete files and folders, inspect the folders and files, open and close the files and folders, and to write and read the files.

The whole project is worth 100 points distributed evenly among four steps; i.e., you will get 25 points for completing each of of the steps.

Take the Quiz Again

Attempt History

	Attempt	Time	Score
LATEST	Attempt 1	39,462 minutes	0 out of 100 *

* Some questions not yet graded

ⓘ Correct answers are hidden.

Score for this attempt: **0** out of 100 *
Submitted Nov 26 at 11:59pm
This attempt took 39,462 minutes.

Step 1

25 points

The file **simfs.zip** contains declarations of the data structures and functions that you must use in this assignment.

The file system volume consists of three parts:

- the superblock,
- a bit vector for maintaining the free storage on the volume, and
- the storage comprised of blocks.

The seed code defines sizes of all the components of the volume.

The superblock holds information about the file system such as the number of blocks, the size of a block, and the reference to the block holding the root folder of the file system.

The bitvector that spans a number of blocks holds bits that indicate whether the corresponding blocks in the volume are free or not; hence, there are as many bits in the bitvector as there are blocks in the storage. You will need to use fast bit-wise operations to search and modify bits in the bitvector.

The storage consists of blocks that are used by folders and files. Each block can hold either a folder or a file descriptor, an index to other blocks, or raw data (file content in bytes).

A file-descriptor node holds meta-data information about a file or a directory such as its size, access rights, creation time, access time, and modification time, along with a pointer to the actual content of the file or the directory. The superblock contains a reference to the block holding a file descriptor for the root folder of the volume.

In a block with the type of a folder, the size field indicates the number of files in the folder, and the block reference field points to the index block that holds references to all files or folders in this folder. Each reference is an index to the block holding the file descriptor of the corresponding file or a folder. Of course, each of the subfolders may in turn hold other files or folders, and so on. There is an upper limit on the number of files or folders in a single folder.

In case of a block with a type of a file, the size field in the file descriptor indicates the actual size of the file. The block reference either points to a single data block if the size of the file is less than the size of a block (less the space needed for the type), or to an index block holding references to data blocks for larger files.

In this step, implement functions to:

- create the file system,
- create a file or a folder,
- delete a file or a folder, and
- obtain file information.

The files are predefined for you in the code with extensive descriptions.

Step 2

25 points

In the second step of the project, you will implement mounting of the simulated file system. The supporting data structures and the mounting function are predefined and described with details in the seed code.

Mounting is a two-stage process of attaching a file system to the existing mounting point. In this step, we assume that the simulated file system is the only file system, so the mounting point of the root of the simulated file system is also the root of the overall file system hierarchy.

The first stage of the mounting process involves a traversal of the simulated volume starting in the root and constructing an in-memory directory of all folders and files on the volume. If our volume was not simulated in memory but was in some hardware storage, then it would certainly be much slower than anything that resides in computer memory. Therefore, this step simulates creation of a data structure that is memory-resident and that provides fast access to the information about the content of the file system without resorting to potentially slow volume access.

The information about the folders in files on the volume should be kept in a directory that is implemented as a hash table. A hash of the name of the folder or a file discovered during the traversal of the volume will be used as an index to the hash table. Each entry in the hash table is a head of a conflict-resolution linked list of nodes that keep information about the folders and files whose names hash to the same entry. The information about the folder or the file is copied to the corresponding list node that also includes a reference to the block in the file system holding the file descriptor for this folder or file. Examine the code to get the details.

The second stage of the mounting process involves copying of the bit vector blocks from the simulated volume to memory-resident version. This is done for efficiency, since, again, if our volume was not simulated, then accessing it would be much slower than accessing memory.

Both the directory and the in-memory bitvector are part of the file system context structure.

If a new file is created or deleted, appropriate changes to the directory and to the bit vector must be made in both the memory and on the volume. Proper care must be taken to ensure consistency of the information in the in-memory data structures and on the volume.

To minimize the chance for collisions, the size of the hash table-based directory has been predefined for you to a prime number surpassing (but not by much) the number of blocks in the file system. You can use any hash function (recall COMP151?); you may try these as well:

- http://en.wikipedia.org/wiki/Hash_function (http://en.wikipedia.org/wiki/Hash_function)
  http://burtleburtle.net/bob/hash/doobs.html (http://burtleburtle.net/bob/hash/doobs.html)
  http://www.cse.yorku.ca/~oz/hash.html (http://www.cse.yorku.ca/%7Eoz/hash.html)

Step 3

25 points

In this step, you will be working on opening a file or a folder, reading from a file, writing to a file, and closing a file or a folder.

In particular, you should implement:

1. global open file table
2. per process file table
3. open() function
4. read() function
5. write() function
6. close() function

All required data structures and functions are predefined in the code.

The open file tables are maintained to provide fast in-memory access to information that resides on potentially slow volume. The global open file table holds that part of the information about the file that is shared among all processes accessing the file. The local open file table is part of the process control blocks that - in this project - is created for the process in the global list of process control blocks as needed. The table holds the information pertinent to the process (such as access rights). Process control block also includes the process identifier, the current working directory for the process, and the number of process open files. Both the global open file table and the process control list are part of the file system context structure that also contains the directory and the bitvector.

When a file is opened, an entry for the file must be created in the global open file table; if it already exists, then the reference count must be increased. At the same time, an entry for the process-specific information must be created in the local open file table.The per-process open file table is part of the process control block for the process. Process control blocks in this project are added and removed from to the global list of process control blocks as needed. The details are in the commentary section of the code.

Closing a file removes the entry for the process from the local open file table and decreases the reference count in the global open file table. If the reference count is zero, the the entry for the file in the global open file table is also removed. If a process does not have any more open files, then its process control block is removed from the global list of process control blocks.

The reading and writing functions in this project write and read files in their entirety. The write function completely deletes the old content and then acquires new blocks

to hold the new one. The details of all functions are in the comment sections of the code.

Step 4

25 points

The last step in the project is to integrate your simulated file system with **FUSE (https://github.com/libfuse/libfuse)** (please click on the link and read the page before going any further).

You must use Ubuntu for this lab.

You may need to install the FUSE library:

```
$ sudo apt-get install libfuse-dev
```

Download the `hellofs.zip` archive with the code implementing a simple file system using FUSE. To get a good feel for what FUSE provides, read **the corresponding very brief tutorial**, compile the code, install the sample file system, and test the system using the file system system interface through the command line interface.

Then, integrate all your work on the simulated file system with FUSE. The cmake configuration file distributed with the seed code for the project already combines the code with the FUSE library.

The high-level API is described best in the source code of **fuse.h (https://github.com/libfuse/libfuse/blob/579c3b03f57856e369fd6db2226b77aba63b59ff/include/fuse.h#L102-L577)**.

For debugging, it is best to run FUSE in a foreground ("-f") single-threaded ("-s") debug ("-d") mode as follows:

```
$ mkdir mnt
$ bin/simfs -f -s -d mnt
```

You will see a lot of debugging information printed in the terminal in which the program is running. You will need another terminal to interact with the simulated file system. Each time you do that, you will see more debugging information printed. Tracing what's going on will help you figuring out what might be incorrect or missing.

Additional resources

A slightly different version of the tutorial can be found here:

https://engineering.facile.it/blog/eng/write-filesystem-fuse (https://engineering.facile.it/blog/eng/write-filesystem-fuse/) .

The following is a very informative more extensive tutorial that redirects all requests to a local file system:

http://www.cs.nmsu.edu/~pfeiffer/fuse-tutorial/ (http://www.cs.nmsu.edu/%7Epfeiffer/fuse-tutorial/)

You may also find studying `fusexmp.c` interesting:

http://lucasvr.gobolinux.org/etc/fusexmp.c (http://lucasvr.gobolinux.org/etc/fusexmp.c)

as it implements a FUSE file system using the standard VFS interface.

Here is a link to the page with FUSE narration (http://www.cs.hmc.edu/%7Egeoff/classes/hmc.cs135.201109/homework/fuse/fuse_doc.html) maintained by prof. Geoff Kuenning (http://www.cs.hmc.edu/%7Egeoff) at Harvey Mudd College. You can find in there many interesting details including how to debug the file system code (how to run the file system in gdb) and also how to get the process identifier and the user of the process making a request by querying the fuse context:

```
pid_t process_id = fuse_get_context().pid;
```

Unanswered

Question 1 Not yet graded / 100 pts

Submission

Please submit all source code for your implementation along with the build configuration files. You must provide transcripts of thorough testing sessions that prove that your code works as specified in the assignment. You must provide tests for creating and deleting files, mounting the file system, opening and closing files, and writing and reading file content. If you have integrated the simulated file system with FUSE, the transcipts should include the interaction with your mounted file system using the Linux file system interface. In case you have not managed to integrate the file system with FUSE, you must provide transcripts of the tests using your own test driver.

Quiz Score: **0** out of 100