Project 01

Due Nov 25, 2020 at 11:59pm **Points** 100 **Questions** 4
Available Nov 5, 2020 at 12am - Dec 12, 2020 at 11:59pm about 1 month
Time Limit None **Allowed Attempts** Unlimited

Instructions

Introduction

In the next several class sittings you will be working on implementing a file system that we will call `cifs`. This file system will work with an actual block device (e.g., a thumb drive), a simulated block device, or a volume contained in a file. The description of all the ways to use `cifs` is embedded in the code.

You will then integrate your implementation of `cifs` with **FUSE** **(http://fuse.sourceforge.net/)** to allow any user of the system to interact with `cifs` volumes using the Linux file system interface, just like with any other file system installed in the operating system. You will be able to navigate the file system hierarchy, create and delete files and folders, inspect folders and files, and write to and read from 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.

There is a substantial seed code provided in the archive file **`cifs.zip`**. You need to follow the lab instructions literally and use the predefined data structures, the interface (functions) and the functionality already implemented in the seed code. The functions to write and read blocks to/from the physical device are implemented in the seed code for `cifs`, but there is also a much simpler standalone application `blockVolume` included for focused experimentation with interacting with block devices.

NOTE 1: To fully understand the descriptions of the project tasks it is necessary to read them along with the comments embedded in the seed code.

All tests done before the integration with FUSE in step 4 must be handcrafted with simulated values that are needed such as process id, user id, access control lists, etc. Conveniently, a simulated FUSE context that is included in the sample code has exactly the same structure as the actual FUSE context that is accessible after the integration. As the previously simulated data become available in the following steps you must revisit the functions that you implemented in the preceding steps and make appropriate modifications.

Keep all tests in test functions `testStep1()`, `testStep2()`, and `testStep3()` that should be developed progressively without deleting the earlier tests. Since all data structures are predefined, you should not have problems with compilation of all successive tests.

Tests for step 4 must be done through the operating system shell, so you will need to capture all testing in a session transcript and include it in the submission.

You must submit incremental solutions for each of the steps. Submissions that include the functionality from the further steps will be rejected.

NOTE 2: You must use Ubuntu Linux for this lab. The `cmake` configuration file assumes access to the developer version of the FUSE library, so you need to install it to allow the code to build:

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

NOTE 3: You must not remove the `CMake` subdirectory of the project, since it contains `cmake` script that finds the `FUSE` libraries in your system. The seed code will not build without access to the libraries.

This quiz is no longer available as the course has been concluded.

Attempt History

	Attempt	Time	Score
KEPT	Attempt 2	13,376 minutes	70 out of 100
LATEST	Attempt 2	13,376 minutes	70 out of 100
	Attempt 1	15,868 minutes	0 out of 100 *

* Some questions not yet graded

ⓘ Correct answers are hidden.

Score for this attempt: **70** out of 100
Submitted Nov 25, 2020 at 6:13pm
This attempt took 13,376 minutes.

Question 1 Not yet graded / 25 pts

Step 1

Description | Implementation | Submission

Description

A `cifs` volume consists of a large number of blocks of equal size that can hold a variety of information. It is divided into three parts:

- ### Bitvector

  The first group of blocks is reserved for a bitvector for managing the free storage on the volume. The bitvector that spans a number of blocks holds bits that indicate whether the corresponding blocks in the volume are free or not. The number of the blocks reserved for the bitvector is determined by the size of the volume, since there must be as many bits in the bitvector as there are blocks in the storage. You are required to use fast bit-wise operations to search and modify bits in the bitvector.

- ### Superblock

  The block following the bitvector is reserved for the volume's superblock. 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.

- ### Storage

  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 as bytes).

  The first two storage blocks (a folder descriptor and an index block) are allocated to hold the content of the the root folder of the file system.

All blocks reserved for the bitvector, superblock are marked as unavailable in the bitvector. All other blocks - including the blocks that hold information for the root folder - are marked as taken when they are allocated.

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 reference to the block with 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 current 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.

BlanchardSeanProject01.zip (https://cilearn.csuci.edu/files/2332430/download)

Unanswered

Question 2

Not yet graded / 25 pts

Step 2

Description | Submission

Description

In this step, 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 multi-stage process:

1. The superblock is read from the volume and copied to the file system context for fast access.
2. The second stage of the mounting process involves copying of the bit vector blocks from the simulated volume to memory-resident version. This is also done also for efficiency, since accessing hardware-based bitvector would be much slower than accessing its memory-based copy. However, care must be taken to make sure that any changes to the in-memory version are properly propagated to the hardware for permanent storage.
3. The third 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. The directory is an efficient data structure that provides fast access to the critical information about the content of the file system without resorting to slow access to an external device. The directory should hold the information about the folders and

files, and preserve the file system hierarchy. For even more efficiency, it should be 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 (the index will be used as the file handle for any future fast reference). Each entry in the hash table is a head of a conflict-resolution linked list of nodes that keep the information about the folders and files whose names hash to the same entry; the file unique global identifier should be used for resolution of hashing conflicts.

The node of the conflict resolution list includes a copy of the corresponding file descriptor block in the file system. To ensure efficient traversal of the file system, the structure of the file system hierarchy is preserved in the in-memory directory by including a file handle of the folder containing the currently analyzed folder or file (i.e., the index to the hash table of the parent folder). Examine the code to get the details.

The in-memory versions of the superblock, the bitvector, and the directory are part of the file system context structure.

If a new file is created or deleted, appropriate changes to the superblock, 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. The seed code implements the djb2 hashing (**http://www.cse.yorku.ca/~oz/hash.html (http://www.cse.yorku.ca/%7Eoz/hash.html)**).

NOTES:

In this step:

- You must modify as needed the functions implemented in step 1. For example, the functions to create and to delete files must also update the in-memory directory.
- If you need some specific data for testing, like process id, user id, or access rights, then use the simulated FUSE context.

Unanswered

Question 3

Not yet graded / 25 pts

Step 3

Description | Submission

Description

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

In particular, you should implement functions to:

1. open a file or a folder,
2. close a file or a folder,
3. read the content of an opened file or a folder,
4. write content to an opened file or a folder.

Files must be opened to perform reading and writing operations. Similarily, a folder must be opened to create or delete files or folders that it contains.

The file system context includes the head of a list of all processes with currently opened files that must be properly maintained to ensure that:

- files are not re-opened before closing,
- access rights are observed, and
- files or folders are not deleted while being still in use (with the use of reference counts maintained in the directory).

The list also maintains a working directory for a process accessing the file system. The list has its head in the file system context which is initialized to `NULL` on the mounting of the volume.

The reading and writing functions in this project write and read files in their entirety. Specifically, the write function first deletes the old content, and then acquires blocks to hold the new data.

The details of the functions are in the comment sections of the code.

NOTES:

In this step:

- You must modify as needed the functions implemented in steps 1 and 2 to accommodate the new functionality.
- If you need some specific data for testing, like process id, user id, or access rights, then use the simulated FUSE context.

Unanswered

Question 4

Not yet graded / 25 pts

Step 4

Description | Implementation | Building | Debugging | Additional Resources | Submission

Description

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).

Some reminders:

- You must use Ubuntu for this lab.
- You need to install the FUSE library:

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

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

Quiz Score: **70** out of 100

This quiz score has been manually adjusted by +70.0 points.