Assignment: Provide the code for a very basic file system which can be mounted using fuse in linux. Use C (highly recommended) or C++. The file system does not need to be large, but should simulate the problems related to working with block devices. We must support directories and files of various sizes up to a reasonable size.

At a minimum, you must support the following functions:

* create
* destroy
* getattr
* init
* link
* mkdir
* open
* opendir
* read
* readdir
* readlink
* release
* releasedir
* rename
* statfs
* unlink
* write

The following are provided for your support:

FUSE Documentation:

<http://fuse.sourceforge.net/>

<http://fuse.sourceforge.net/doxygen/index.html>

FUSE function reference:

<http://fuse.sourceforge.net/doxygen/structfuse__operations.html>

Examples and Tutorials:

<https://code.google.com/p/fuse-examplefs/>

Fusekit for C++:

<https://code.google.com/p/fusekit/> (SEE trunk/example for examples; untested!)

Malloc/free primer:

<http://en.wikipedia.org/wiki/C_dynamic_memory_allocation>

Fopen,fread,fclose primer:

<http://www-control.eng.cam.ac.uk/~pcr20/www.cppreference.com/stdio_details.html>

<http://www.thegeekstuff.com/2012/07/c-file-handling/>

Limitations: Which the file system will not be used extensively, it should be able to support, at least, a few different files with varying sizes. Assume the following limits on the file system, however these may change in the future:

|  |  |
| --- | --- |
| Maximum number of blocks | 10,000 |
| Maximum file size | 1,638,400 bytes |
| Block size | 4096 bytes |

Prior assumptions: You must have access to a Linux machine. You do not need root access to the system and I would recommend that you have this in a VM so that it is easier to access and restart. Use virtualbox and install Ubuntu 14.04LTS for this.

Step 1)

Read this document fully and all the associated primer’s above. Compile and test the hello.c filesystem in the FUSE website to make sure you know that your system works and understand the basics of fuse.

Step 2)

The logical layout of our file system will be hierarchical, however we must understand the limitations of the block device

Our layout will consist of a given number of blocks, each existing as a single file on the “host” file system with the name “fusedata.X” where X is a numeric value from zero to one less than the maximum number of blocks. Each block is preallocated (upon filesystem creation) with all zeros and stored on the host file system. This means, that upon creation of the filesystem we will create 10,000 files each of 4096 bytes on the host file system.

All structures on our file system will be contained within {}. The values will be in a form of variable:value wherever possible.

Since we will need to store more than just plain files, we will use block (file) zero as the “super” block. The super block will contain any information we wish to make persistent about the file system including its creation time, number of times it was mounted, the device\_id (always 20), the location of the start and end of the free block list, the location of the root directory block and the maximum blocks in the system. The super block will look like the following:

{creationTime: 1376483073, mounted: 50, devId:20, freeStart:1, freeEnd:25, root:26, maxBlocks:10000}

You must start by creating the above superblock and storing it in fusedata.0 on the host filesystem. We also need to create the free block list which you may assume is an array of numbers indicating the available blocks. Since we have a limited storage in each block, you should evenly distribute the list of free blocks among all of the available blocks in the free block list. For example, block 1 (fusedata.1; the first in the free block list) would, initially, contain an array of the numeric values of blocks starting from 27 and continuing up to and including 399 (373 entries on the array). Block 2 would contain an array of 400 blocks starting from 400 and continuing to 799. Yes, this means that we must aggregate the 25 blocks to find all the free blocks, but that’s not unusual for a block device. You may store this list as a simple comma separated value

Step 3)

Since we are simulating a real block device here, we need to store directory information for each directory. The directory will fit into one block (in a real block device we would consider the possibility that the directory exceeded the block size, but let’s not add too much complexity). We can trust that each directory will have the following format:

{size:1033, uid:1000, gid:1000, mode:16877, atime:1323630836, ctime:1323630836, mtime:1323630836, linkcount:4, filename\_to\_inode\_dict: {f:foo:1234, d:.:102, d:..:10, f:bar:2245}}

The filename\_to\_inode\_dict is a file table which is broken down into three parts, separated by colons “:”. The first character of the entry in the filename\_to\_inode\_dict will indicate that it is a file “f’ directory “d” or special “s” (but don’t worry about s for now). This letter is NOT part of the filename, just a descriptor to tell you what type of inode to expect.

Now it’s time to create the root directory inode

{size:0, uid:1, gid:1, mode:16877, atime:1323630836, ctime:1323630836, mtime:1323630836, linkcount:2, filename\_to\_inode\_dict: {d:.:26,d:..:26}}

Set . and .. to the value in the superblock and write it out.

Each time a new directory is created, you must allocate a new block (take the first available block from the free block list), store the directory information in the format, and add the new block to the parent’s filename\_to\_inode\_dict.

Step 4)

Files need to be handled carefully in the file system. The format for a file’s inode information will be:   
{size:1033, uid:1, gid:1, mode:33261, linkcount:2, atime:1323630836, ctime:1323630836, mtime:1323630836, indirect:0 location:2444}

This data will be stored in the inode location referenced in the filename\_to\_inode\_dict entry in the parent directory. When a file is created, we must consider how large it is. Since our file system can only handle blocks of 4096 bytes or less, we have to use a level of indirection if we want to store a file larger than that size. Should we need to store a file of between 4097 and 1,638,400 bytes, we need to use an index block. The index block is simply an array of pointers to other blocks. Since we will not be handling anything larger than 1,638,400 bytes, and 400 pointers will easily fit into an index block, we need only a single level of indirection.

We will use the “indirect” field in the inode table to indicate if the “location” field is referencing file data or an index block.