LLFS: The *Little* Log File System!
CSC 360 2019

Goals of this assignment

1. Manipulate a simulated disk, and format that disk with your version of the Little Log File System (LLFS) [20 marks].

2. Modify LLFS to support:
   a. reading and writing of files in the root directory [20 marks]
   b. creation of sub-directories [10 marks]
   c. reading and writing of files in any directory [15 marks]
   d. deletion of files and directories [10 marks]

3. Make LLFS robust, so that the file system is less likely to be corrupted if the machine it is running on crashes [15 marks].

4. Submit test files used to exercise functionality [10 marks].

Specifics of the code

In a directory called */disk*, you will create a virtual disk environment, and in a directory called */io*, you will create a program called *File.c*. This will be the basis for your filesystem.

In a directory called */apps,* build your test code as *apps/test01.c* to test simple functionality, such as see how blocks are read/written to a disk. I/O happens in one-block chunks. Write code in *apps/test02.c* to demonstrate how disks are created and how disks are mounted.

General ideas and instructions

The code in */disk* will act as a library for a simulated disk, and your version of LLFS will be built using this library. The disk library will just use a simple file, named "vdisk" in the directory from which you run your test programs. This file should be about 2 megabytes in size.

The contents of *vdisk* are wiped clean when the disk is explicitly formatted via a call to *InitLLFS*. The idea is that to treat this file as though it were a raw disk – you can run a program which uses LLFS to write a file to the simulated disk, and then run another program that reads the file. Of course, the same program may write and read (or even delete) the file. Deleting *vdisk* from your directory allows you to start a simulated disk from scratch.

One of the challenges in building LLFS is to design and implement the set of data structures that map file names onto disk blocks. For this mapping, you will use a per file *inode*. Inodes map a file (a collection of bytes) to the disk representation of a file (a collection of disk blocks). Inodes thus contain what is referred to as the file's *metadata,* and as such contain mappings to disk blocks, or to *indirect blocks*. More on indirect blocks later in class.

Another challenge is be able to allocate blocks correctly. You need to track unused or free blocks. For this you can use a simple data structure, such as a bit map, which just tracks

which blocks are allocated, and which are free.

Another challenge will be to organize the naming structure into a directory tree. Directories are mappings of names to *inodes* for all files and directories contained within the directory. This might not seem like much of a challenge in itself, but handling this data structure in a way that makes LLFS robust requires some thought.

An interesting feature of these data structures is that they requires *persistence* (meaning they has to be stored on disk, as well as being manipulated in memory). The issue of making LLFS robust revolves around making an effort to keep their on disk copies up to date so that, if the system crashes, they will survive the crash without becoming inconsistent (that is, without misplacing any blocks, or having allocated blocks showing up as free, etc).

Goal 1: Create and manipulate the simulated disk, and format that disk with your version of the Little Log File System (LLFS).
Here, you will need to think about how blocks are read from and written to a disk, in one-block chunks, and how disks are created and file systems mounted.

Some Specifications for LLFS on disk:
This list is not exhaustive with respect to everything you will need, as you will be designing parts of LLFS based on your own decisions, but it will be a start!

Disk parameters
· Size of a block: 512 bytes
· Number of blocks on disk: 4096
· Name of file simulating disk: “vdisk” in current directory
· Blocks are numbered from 0 to 4095

Block 0 – superblock
· Contains information about the filesystem implementation, such as
· first 4 bytes: magic number
· next 4 bytes: number of blocks on disk
· next 4 bytes: number of inodes for disk

Block 1 – free block vector
· With 512 bytes in this block, and 8 bits per byte, our free-block vector may hold 4096 bits.
· First ten blocks (0 through 9) are not available for data.
· To indicate an available block, bit must be set to 1.

Other Blocks
· You can reserve other blocks for other persistent data structures in LLFS
· One thing to consider is how you are going to keep track of there all the inodes in the filesystem.
· Each inode has a unique id number.
· Each inode is 32 bytes long.
· An inode has to be allocated to represent information for the root directory.

Goal 2: Modify the system to support basic LLFS functionality.
The first challenge is to create a file (i.e., opening a file for "write access" that does not exist in a directory is the same as creating file) and then to ensure bytes are written to correctly regardless if they are within a block or span blocks. Converting from file position to a block and offset is use division and modulo arithmetic.

Basic steps you need to take to create a file include allocating and initializing an inode for the file, writing the mapping between the inode and the file name in the directory, and making this information persistent.

Since LLFS is a log-structured file system[1], writing to a file involves writing to a log, instead of updating disk blocks in place. For example, imagine you have a file, *csc360midterm.doc* that you want to change. Say the inode for this file contains pointers to 4 data blocks, at disk block locations 42, 43, 56 and 57. Note that the blocks are not contiguous on disk, meaning the disk will have to seek to block 42, read 2 blocks, then seek again to 56, and read another 2 blocks. If you change data in the 3rd block, LLFS will not modify that block. Instead, it will store a copy of the file's inode and the 3rd block to a growing part of the file system called the *log*. Every now and then, LLFS must checkpoint and make things persistent, such as updating the on-disk list of inodes, as well as freeing the "unused" parts of the files (i.e., the old inode, and that 3rd block of the file).

Writing to a file in LLFS thus requires loading the file's associated inode into memory, allocating disk blocks (marking them as allocated in the freelist), modifying the file's inode to point to these blocks, and writing the data to these blocks. Releasing inodes involves removing pointers to the disk blocks from the inode, and marking the disk blocks as free. Modifying directories is very similar. Directories are just files with a specific format. To keep things simple, we will limit the depth in LLFS to 4.

To get you started, consider the following format for inodes and directories:

Inode format:
· Each inode is 32 bytes long
· First 4 bytes: size of file in bytes; an integer
· Next 4 bytes: flags – i.e., type of file (flat or directory); an integer
· Next 2 bytes, multiplied by 10: block numbers for file's first ten blocks
· Next 2 bytes: single-indirect block
· Last 2 bytes: double-indirect block

Directory format:
· Each directory block contains 16 entries.
· Each entry is 32 bytes long
· First byte indicates the inode (value of 0 means no entry)
· Next 31 bytes are for the filename, terminated with a "null" character.

Note re testing how LLFS is doing with respect to reading and writing files: You can write a string to a file (many times to fill a larger file!), and examine the results of a write by examining

the contents of "vdisk" using *hexdump –C* (check it out!). In a similar way you can also read back the data from a file and compare it against what should be in the file.

Goal 3: Making LLFS Robust

By using the disk, we can make data and metadata persistent. In LLFS, this means we need to checkpoint changes from time to time. For this part of the assignment, you have to design and implement a policy for checkpointing and recovering. You will need to balance the need to constantly update disk, with the expense of disk updates.

In particular, you want to think about how to handle two scenarios: crashes that occur after free blocks have been allocated to a file, and crashes that occur just after blocks have been removed from the freelist. The core question is: can you make your version of LLFS survive these crashes and not corrupt the system?

Note: during reboot, UNIX machines run a program called "fsck" (file system check) that does a walk through of file system data structures, repairing if need be. Feel free to use this approach.

Simplifying Assumptions

Only one thread will ever manipulate files. Of course, that thread may have several files open. This simplifies LLFS as synchronization won't be required.

All pathnames used for Open, Mkdir, Rmdir and Unlink are absolute (i.e., begin with a leading "/").

Several support functions for working with bit vectors, examining and modifying byte arrays, etc. will be useful to have around!

Directories can only be created one "level" at a time. As an example, if a user wishes to create a new directory with the pathname "/home/csc360/foo" and only "/home" exists, then two separate "mkdir" requests are required: one to create "/home/csc360", and a second to create "/home/csc360/foo". Similar reasoning applies to the removal of directories.

A directory may be deleted only if it is empty (i.e., all files and sub-directories are already deleted).

What you must submit

- Your version of `/io/File.c`
- Test files you have developed in `apps`.
- The `Makefile` from the `apps` directory.
- A very short (one page) document describing your design and implementation decisions and tradeoffs in a file named `report.txt`.