Log-Structured File System Project Report for Phase 2

CSc 552: Advanced Operating Systems

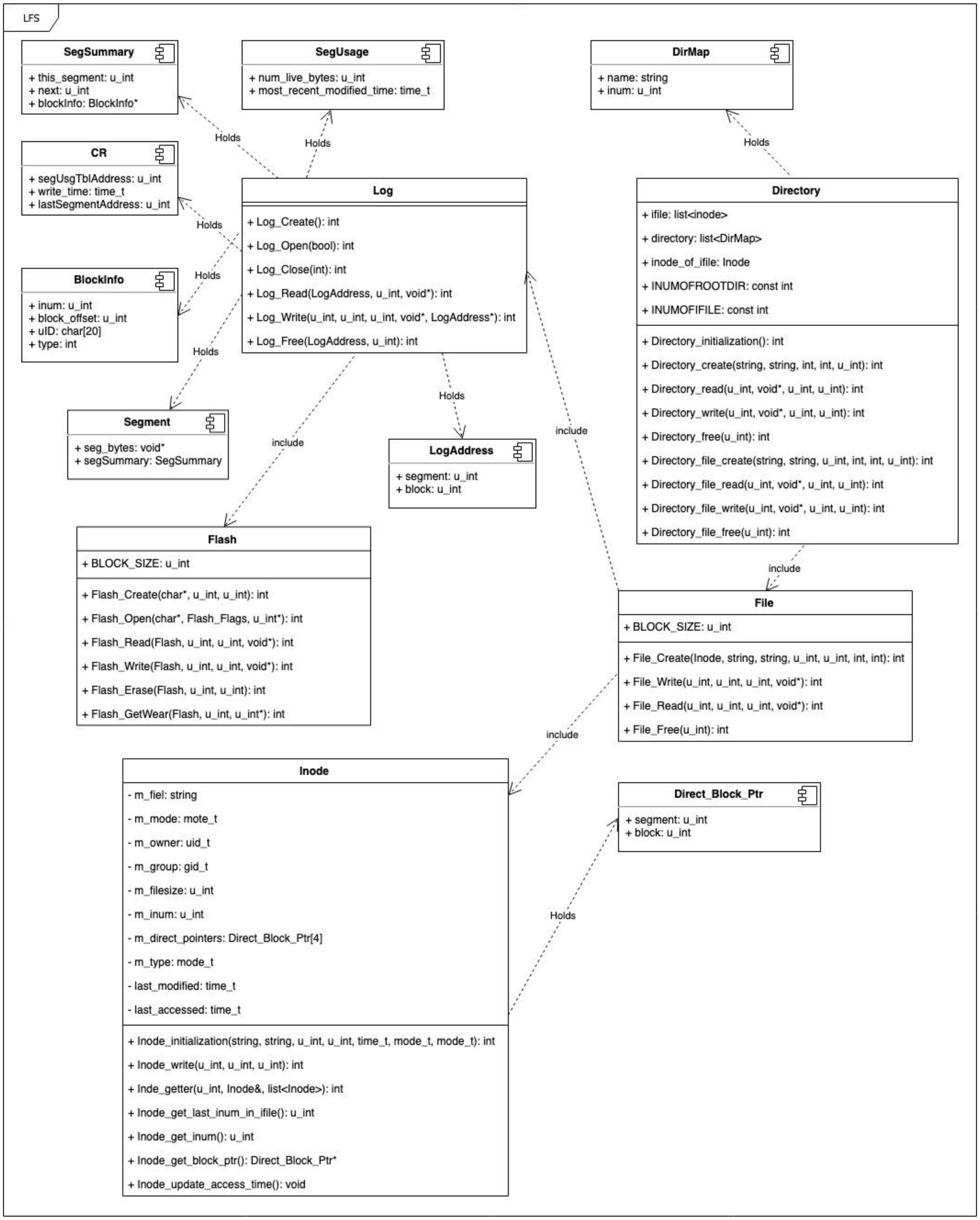
Deadline: May 1, 2019 11:59 PM

Team Members

Sabin Devkota

Terrence Lim

UML diagram of our LFS implementation


LFS
SegSummary
+ this_segment: u_int
+ next: u_int
+ blockInfo: BlockInfo*
SegUsage
+ num_live_bytes: u_int
+ most_recent_modified_time: time_t
DirMap
+ name: string
+ inum: u_int
CR
+ segUsgTblAddress: u_int
+ write_time: time_t
+ lastSegmentAddress: u_int
BlockInfo
+ inum: u_int
+ block_offset: u_int
+ uID: char[20]
+ type: int
Segment
+ seg_bytes: void*
+ segSummary: SegSummary
Log
+ Log_Create(): int
+ Log_Open(bool): int
+ Log_Close(int): int
+ Log_Read(LogAddress, u_int, void*): int
+ Log_Write(u_int, u_int, u_int, void*, LogAddress*): int
+ Log_Free(LogAddress, u_int): int
LogAddress
+ segment: u_int
+ block: u_int
Directory
+ ifile: list<inode>
+ directory: list<DirMap>
+ inode_of_ifile: Inode
+ INUMOFROOTDIR: const int
+ INUMOFIFILE: const int
+ Directory_initialization(): int
+ Directory_create(string, string, int, int, u_int): int
+ Directory_read(u_int, void*, u_int, u_int): int
+ Directory_write(u_int, void*, u_int, u_int): int
+ Directory_free(u_int): int
+ Directory_file_create(string, string, u_int, int, int, u_int): int
+ Directory_file_read(u_int, void*, u_int, u_int): int
+ Directory_file_write(u_int, void*, u_int, u_int): int
+ Directory_file_free(u_int): int
Flash
+ BLOCK_SIZE: u_int
+ Flash_Create(char*, u_int, u_int): int
+ Flash_Open(char*, Flash_Flags, u_int*): int
+ Flash_Read(Flash, u_int, u_int, void*): int
+ Flash_Write(Flash, u_int, u_int, void*): int
+ Flash_Erase(Flash, u_int, u_int): int
+ Flash_GetWear(Flash, u_int, u_int*): int
File
+ BLOCK_SIZE: u_int
+ File_Create(Inode, string, string, u_int, u_int, int, int): int
+ File_Write(u_int, u_int, u_int, void*): int
+ File_Read(u_int, u_int, u_int, void*): int
+ File_Free(u_int): int
Inode
- m_fiel: string
- m_mode: mote_t
- m_owner: uid_t
- m_group: gid_t
- m_filesize: u_int
- m_inum: u_int
- m_direct_pointers: Direct_Block_Ptr[4]
- m_type: mode_t
- last_modified: time_t
- last_accessed: time_t
+ Inode_initialization(string, string, u_int, u_int, time_t, mode_t, mode_t): int
+ Inode_write(u_int, u_int, u_int): int
+ Inde_getter(u_int, Inode&, list<Inode>): int
+ Inode_get_last_inum_in_ifile(): u_int
+ Inode_get_inum(): u_int
+ Inode_get_block_ptr(): Direct_Block_Ptr*
+ Inode_update_access_time(): void
Direct_Block_Ptr
+ segment: u_int
+ block: u_int
Holds
Holds
Holds
Holds
Holds
Holds
Holds
Holds
Holds
include
include
include
include

Introduction

Work Distribution

Layers	Implementer
LFS (lfs.c)	Sabin Devkota
Mklfs (mklfs.c)	Sabin Devkota
Log (log.h, log.c)	Sabin Devkota
Directory (directory.h, directory.cpp)	Terrence Lim
File (file.h, file.cpp)	Terrence Lim
Fuse (fuse_implementation.cpp)	Terrence Lim
Inode (inode.h, inode.cpp)	Terrence Lim

Status

Currently, the code for the log , file, and directory layers are implemented. We have tested the code for these layers. We are still working on getting the FUSE integration to work. All code for FUSE integration are written. Majority of the the codes for Phase 2 are written. However, we have failed to implement "remove directory", "remove file", "links" and "segment cleaner" which is partially written. The code for segment cleaning is still a work in progress because we worked on getting the FUSE integration to work before finishing the segment cleaner.

Organization

Thoeretically, if our code was running as expected, the LFS uses Fuse to initialize and call each functions from the directory layer. Directory leyer calls file layer to access log layer and inodes class to create/read/modify them.

Result

Unfortunately, our group has failed to successfully get the program running, though, each layers' at least the basic functionalities are implemented. The reason of failure is due to the problems with integration and unexpected time spent in debugging. For the demo, we will prepare for each layer's unit test in a way that they were supposed to work if the Fuse integration was successful.

Description of layers

Log Layer

We have implemented a prototype log layer. The log layer is responsible for reading and writing to the log. It sits on top of the Flash layer and performs all the operations related to creating and formatting the flash file, and reading/writing to log. In addition, it maintains segment summary information for each block in the segment, writes checkpoint information and loads from checkpoint when mounting, and puts the checkpoint region's information inside the superblock. The checkpoint region is stored in the log itself.

We describe the important data structures and functions used by the Log Layer in our prototype implementation below.

Data Structures:

SuperBlock: This structure contains the size of segments in block, size of block in sector, number of segments, wearlimit, and the addresses of the checkpoint region in the log. It is stored in the first segment of the log.
SegUsgTbl: The segment usage table stores the number of live blocks in the segment as well as the most recently modified time of any block in the segment.
CR: CR stores the checkpoint regions for the log. It contains the inode of the ifile, address of the segment usage table, time of creating the checkpoint, and last segment written when creating the checkpoint. Currently, we use two checkpoint regions and alternate between the two to store the checkpoint information.

SegSummary: The segment summary contains the segment offset of the current segment (in flash) as well as segment offset of the next segment. For each block in the segment, it stores the inum and the block offset of the file it belongs to. In addition, it also stores the type of block which can be either a file data block or Other block. (Other block type is used for storing checkpoint information.)
SegmentCache: The segment Cache stores the tail segment of the log as well as other segments that were read recently. The segToCacheMap finds a segment within the cache. If the segment being read is not in the cache, it is loaded to the cache.

Functions implemen ted:

Log_Create: It creates the flash file with the given number of segments, segment size, and wearlimit. It initializes the data structures required for the log. Then it calls the directory layer to initialize the directory structure.
Log_Open: It opens the flash file to load the superblock and the checkpoint region. It initializes the segment cache with the empty tail segment and initializes the data structures required for the log.
Log_Close: It frees the memory occupied by the segment cache, writes the checkpoint to the flash file and closes the flash file.
Log_Read: It reads the specified bytes from a given log address and writes to the buffer. It uses the segToCacheMap structure to find the segment in the segment cache. If the segment is not in the cache, then it reads the segment from the flash file and loads it to the cache.
Log_Write: It writes the specified bytes from the buffer to the tail segment of the log. It returns the log address of the write back to the calling function.

writeToTail: This function writes the specified bytes to the tail of the log and is used by Log_Write function.
writeTailSegToFlash: This function writes the tail segment to flash when the segment is full. It sets the next empty segment as the new tail segment.
writeCheckpoint: It writes the checkpoint to the log, and the address of the checkpoint region to the superblock. We use two checkpoint regions. It alternates between the two when writing the checkpoint.
loadCheckpoint: It loads the superblock from the flash and loads the checkpoints using the logaddresses found in the superblock.
Log_Free: It frees the blocks in the flash by setting the *isLive* flag to false for the blocks in the segment summary.

Segment Cleaning:

The code for segment cleaning is a work in progress. The following are the major functions related to segment cleaning.

SetBlockDead: This function is used by *Log_Free* to set the *isLive* flag to false in the segment summary when the block is freed by *Log_Free.*

isLive: The isLive function checks if a block is live. It first checks if the *isLive* flag is set to false in the segment summary for that block. If not, it compares the address in the block offset of the inode of the file containing that block with the address of the block. If they are the same, then the block is live, otherwise it is dead.

computeLiveNess: This function goes through all the used segments in the flash and computes the number of live blocks in each segment by calling the isLive function on each block.

cleanSegments: This function finds the segments that have live blocks below a set threshold. It selects N dirty segments and creates M clean segments (N>M) by copying the blocks from the dirty segments to the clean segments. Then it updates the inode of the file containing the blocks with the new address of the block. It then updates the list of clean segments in the flash.

FUSE in LFS.c:

The *Log_Open* function calls the *fuse_main* function and passes the struct for *fuse_operations* which points to the functions implemented in *fuse_implement.cpp.*

Directory Layer

A directory layer is an intermediate layer between the log layer and file layer, which maintains the directory hierarchy. In our implementation, any access to files are done via the directory layer. For example, the directory layer class holds Directory_file_* functions that calls functions from the File class to access the file and inodes.

Functions implemented:

Directory_initialization: This function initializes the directory layer and creates a root directory (“/”). In order for it to generate the root directory, it calls Directory_create function. If the initialization was done successfully, it will return 0 else return 1 and print out error message.
Directory_create: This function gets called every time when a new directory needs to be created. It creates default files (“.” and “..”) together.
Directory_read: This function will read in the inode of a directory and the directory file.
Directory_write: This function will be invoked to update the inode and the directory file.
Directory_free: This function will free up the memory that the directory (directory file and inode) is occupied when the directory gets deleted.
Directory_chmod: As name infers, this function changes the mode of the directory/file by calling the Inode_Chmod function.
Directory_chown: This function changes the user id and group id with the passed ids by calling Inode_Chown.

Directory_file_create: This function calls File_Create function by passing the empty inode, which will be initiated by the function and returned. If the inode was successfully created, it will push the inode to the global in-memory ifile for later to be inserted into log on disk.
Directory_file_open: This function is to open the file with a given path.
Directory_file_write: This function is responsible calling File_Write with already generated inode (passing inum), to set the direct pointers to the file blocks.
Directory_file_read: This function simply reads the inode via calling the File_Read and fills the buffer that was passed as a parameter.
Directory_file_rename: This function renames the file with a given path by calling File_Rename.
Directory_file_free: This function is for freeing up the memory that the inode of the deleted file is occupying. This function will call the File_Free to perform the operation.
Directory_file_getattr: This function returns the current status of the inode.
Directory_statfs: This function calls File_Statfs to retrieve the current filesystem’s status.

File Layer

A File layer is a layer between the Directory layer and Inode that is responsible for creating, reading, writing, and freeing up the inode. All the inodes that were created within this layer will be returned to directory layer and they'll be stored in the global extern ifile list, so they can be widely accessed (exclusively for Log layer) by all layers across the file system.

Functions implemented:

File_Create: This function is responsible for generating the inode and initialize it to the default values and the passed metadata by calling the Inode_Initialization function.
File_Write: This function is responsible for setting the direct pointers of the inode with a given inum. The function will first call the Log_write to retrieve the log address object from the log layer. If the retrieval was successful, it will look for the target inode based on the given inum and assign each of 4 direct pointers to the first 4 blocks of a file. Then, update the inode that is stored in the ifile with the updated version.
File_Read: This function is responsible for updating the passed empty buffer by calling the Log_Read from the log layer. The log_Write will return the address of a first block. Then, using this address, each 4 block's address locations will be computed within the loop and then set to the direct pointers.

File_Free: This function will be invoked in the directory layer's Directory_File_Free to free up the memory that the inode of a deleted file is occupying.
File_Getattr: This function calls inode layer's Inode_Set_Stbuf to retrieve the current status of the file inode.
File_Statfs: This function retrieves the status of the filesystem by seeking the Superblock values.

Inode

Inode class holds a struct container for inode table object and all the member functions to access and modify the object.

Functions implemented:

Inode_Initialization: This function will initialize the file inode with given metadata. At the initialization stage, uid and gid will be set to the current user and group as a default, and the last modified and access time will be set to current time as a default. The file is empty, so the file size is zero as well.
Inode_Write: This function gets called in the File_Write in the file layer to actually set the inode's direct pointers. Later on in the Phase 2, this function will be added with a functionality to set indirect pointers.
Inode_Getter: This function is simply an inode return function. This will receive the in-memory ifile as a list, then traverse it until it finds the matching inum. If found, the Inode& will be assigned with the found inode and return 0, else return 1.
Inode_Get_Last_Inum: This function will simply return the last inum, so that a new inum can be provided to the new inode sequentially.
Inode_Get_Inum: This function will return the inum of the current inode.
Inode_Get_Block_Ptr: This function will return the direct pointer arrays of inode, so it can be acced in the File layer and be updated as necessary, such as in the File_Write function.

Inode_Update_Access_Time: This function updates the last access time to the file (inode). For example, if a system opens a file via File_Read or later File_Append, etc, this function will be invoked to update the access time.
Inode_Find_Inode: This function is for finding and retrieving the inode when only a path and filename are known.
Inode_Chmod: This function checks the current mode of the inode and modify with the passed mode.
Inode_Chown: This function checks the current owner of the inode and modify with the passed owner id and group id.
Inode_Rename: This function changes the filename of the inode.
Inode_Check_Mode: This function is usually used in in the Inode_Chmode to check the current mode of the file (inode).
Inode_Get_Total_Num_Of_Inodes: This function simply counts the number of inodes in the ifile and return to the caller.
Inode_Set_Stbuf: This function is to set the stbuf's variables with the current status of the inode.

Fuse Implementation

- Our program's fuse functions are implemented under **fuse_implement.h** and **fuse_implement.cpp**. Unfortunately, our group has failed to integrate the layers with the fuse.
- The functions laid out below have code stubs that we believed to should be working.
- All fuse functions call directory layer's Directory_<name> functions.

Functions:

imp_init
imp_file_getattr
imp_file_open
imp_file_read
imp_file_write
imp_access

imp_unlink
imp_statfs
imp_file_release
imp_mkdir
imp_dir_open
imp_dir_read

imp_dir_release
imp_destroy
imp_file_create
imp_rename
imp_chmod
imp_chown

I'm omitting the descriptions for each functions for the fuse implementation as they are really code stubs for calling directory layer functions.