Log-Structured File System Design
for
Phase 1

CSc 552: Advanced Operating Systems

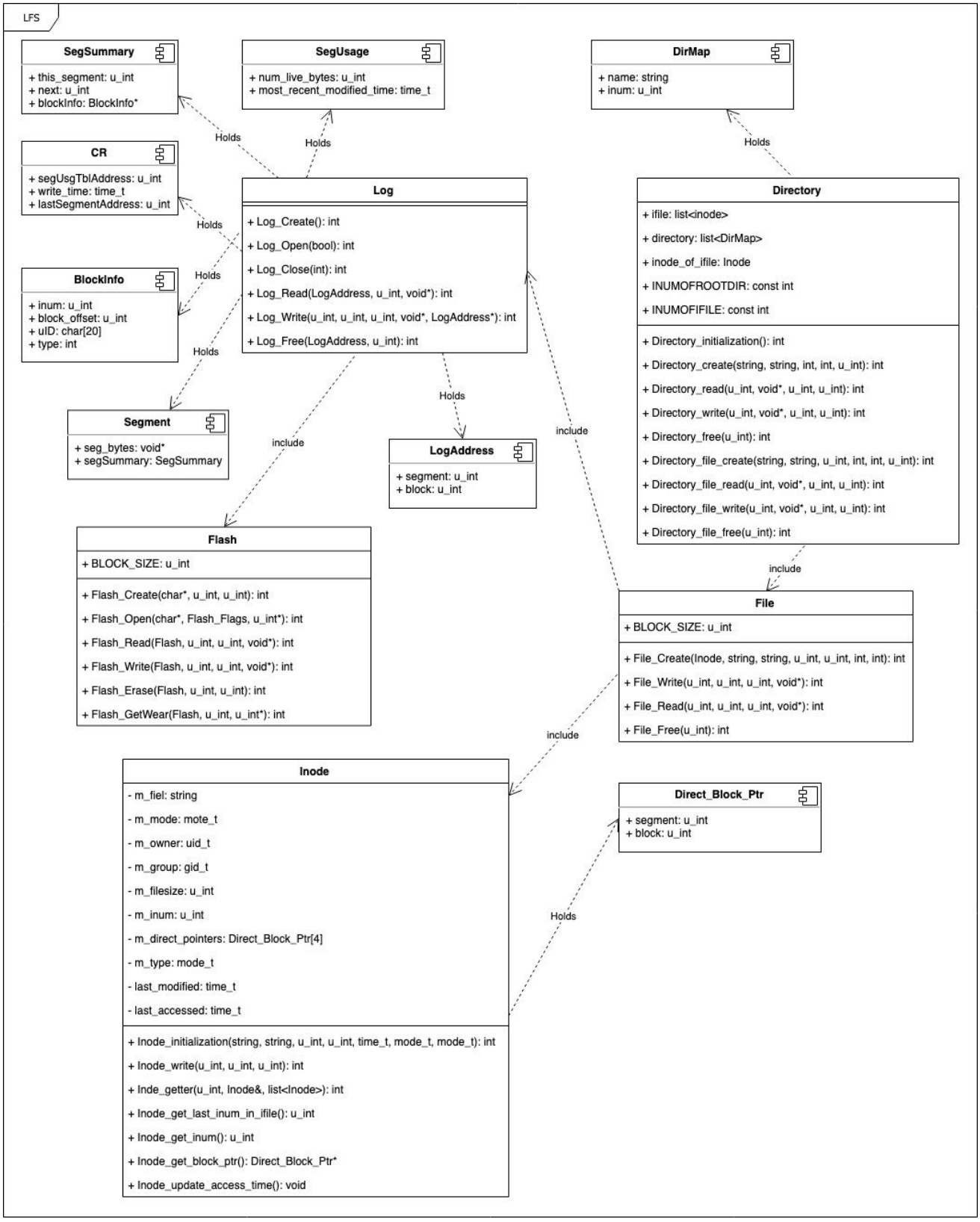
Deadline: March 18, 2019 11:59 PM

Team Members

Sabin Devkota

Terrence Lim

UML diagram of our LFS implementation:



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 bytes 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:

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.

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:

int 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.

int Directory_create(string, string, int, int, u_int): This function gets called every time when a new directory needs to be created. Since a directory is actually a special type of a "file", it calls Directory_file_create, which it will call File_Create function from the File layer to create and initiate a new inode.
Furthermore, as all directories are default to hold two files "." and "..", it calls Directory_file_create function twice more, but this time it adds those two new file names and inum into the in-memory directory <name, inum> structure, which will later be written into a disk in Log layer.

int Directory_read(u_int, void*, u_int, u_int): This function will read in the inode of a directory and the directory file.

int Directory_write(u_int, void*, u_int, u_int): This function will be invoked to update the inode and the directory file.

int Directory_Free(u_int): This function will free up the memory that the directory (directory file and inode) is occupied when the directory gets deleted.

int Directory_file_create(string, string, u_int, int, int, u_int): 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.

int Directory_file_write(u_int, void*, u_int, u_int): This function is responsible calling File_Write with already generated inode (passing inum), to set the direct pointers to the file blocks.

int Directory_file_read(u_int, void*, u_int, u_int): This function simply reads the inode via calling the File_Read and fills the buffer that was passed as a parameter.

int Directory_file_free(u_int): 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.

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:

int File_Create(Inode, string, string, u_int, u_int, int, int): 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.

int File_Write(u_int, u_int, u_int, void*): 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.

int File_Read(u_int, u_int, u_int, void*): 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.

int File_Free(u_int): 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.

Inode

Inode is not really a layer, but a class that holds all the member variables to hold the metadata of inode, and provide access functions.

Functions:

int Inode_Initialization(string, string, u_int, u_int, time_t, mode_t, mode_t): 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.

int Inode_Write(u_int, u_int, u_int): 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.

int Inode_getter(u_int, Inode&, list<Inode>): 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.

u_int Inode_get_last_inum_in_ifile(): This function will simply return the last inum, so that a new inum can be provided to the new inode sequentially.

u_int Inode_get_inum(): This function will return the inum of the current inode.

Direct_Block_Ptr* 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.

void 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.

Status

We have implemented the prototype for log, file and directory layers. We are in the process of testing our implementation. We have not implemented lfsck currently. After testing our layers individually, we will be implementing the FUSE functionality to create a working LFS prototype.