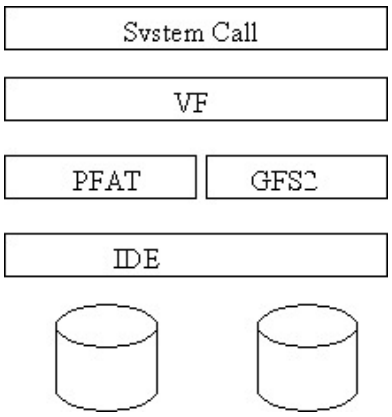
Project 5: GeekOS Log-Structured File System

1 Overview

The purpose of this project is to add a writable filesystem, LFS, to GeekOS. Unlike the existing PFAT, LFS includes directories, inodes, direntries, etc. The new filesystem can be bootable, but will for development reside on the second IDE drive in the emulator. The PFAT drive will continue to hold user programs while you format and test your implementation of LFS.

1.1 VFS Introduction

Since GeekOS will have two types of filesystems (PFAT and LFS), it will use the virtual filesystem layer (VFS) to direct requests to an appropriate filesystem (see figure below). We have provided an implementation of the VFS layer in the file vfs.c. The VFS layer will call the appropriate LFS routines when a file operation refers to a file in LFS.



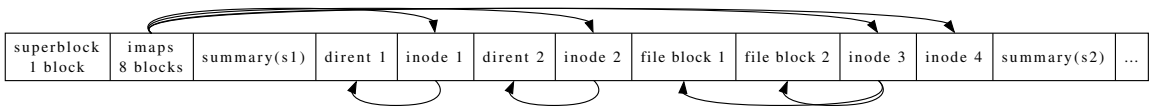
The implementation of PFAT is in pfat.c. You will implement LFS in lfs.c and relevant system calls in syscall.c.

VFS picks the functions to call based on supplied structures containing function pointers, one for each operation. For example, see Init_PFAT in pfat.c: this initializes the PFAT filesystem with VFS by passing it a pointer to s_pfatFilesystemOps, a structure that contains function pointers to PFAT routines for mounting (and formatting) a filesystem. Other PFAT functions are stored in different structures (e.g., look at the PFAT_Open routine, which passes the gs_pfatFileOps structure to VFS). You will analogously use s_lfsFilesystemOps, s_lfsMountPointOps, s_lfsDirOps, and s_lfsFileOps in lfs.c. You should also add a call to Init_LFS provided in lfs.c to main.c to register the LFS filesystem. In general, use the PFAT implementation as your guide to interfacing LFS with VFS.

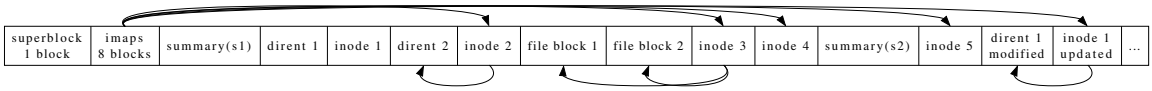
1.2 LFS Filesystem Data

1.2.1 Overview

Here is the over view of the filesystem:



If we simply create an empty file in the root directory:



1.2.2 Superblock (lfs_superblock)

magic: 0x4C474653
version: 0x00000100
block_size 512, 1024, 4096
num_check_points
blocks_per_disk

The superblock contains the file system parameters and is used by the kernel to load and interpret the filesystem. It must be stored at byte offset PFAT_BOOT_RECORD_OFFSET (482) (NOT block offset). You can expect it to be there on mount. If it's not there, the file system is broken and would have to be repaired before being mounted.

You may ignore setupStart, setupSize, kernelStart, and kernelSize, all of which are to support the loading of the kernel when a LFS image is made bootable.

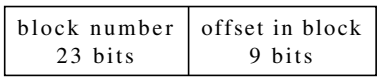
1.2.3 Primitive types

Both primitive types are unsigned integers; the separate data type is meant to ease type checking.

lfs_blocknum The number of the block (not sector) on disk.

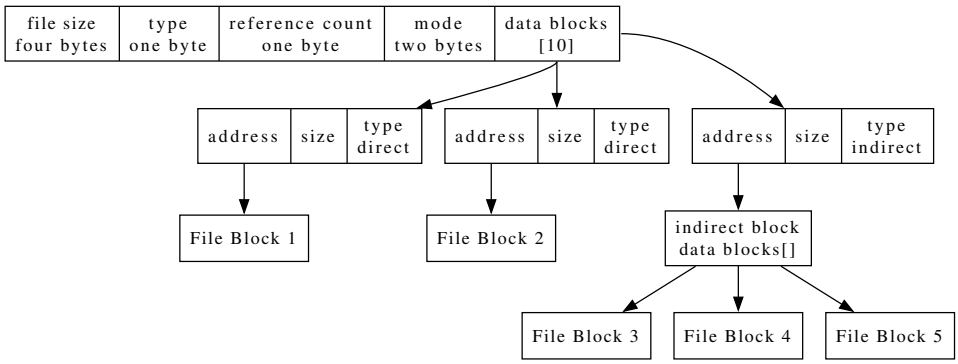
lfs_inodenum The number of the inode (not the block containing the inode). There are a fixed number of inodes, and thus possible files and directories, in a given disk image.

address The address used by this filesystem. This is not standard but for GeekOS and we have provided some macros for you, this is defined to be the following:



1.2.4 Inode (lfs_inode)

Inodes are data structures that store information about a file or directory on the filesystem. Their structure in LFS is as follows:



The size of the inode is constrained so that an even number of inodes fit into each block (inodes won't span blocks). The inode does not identify itself; if you need to manipulate an inode, the number is a necessary identifier; the cached inode itself will not have enough information to write it back to disk in the right place.

The type can be either:

LFS_DIRECTORY It's a directory. It won't be read or written directly by applications.

LFS_FILE It's a file.

The reference count should be zero in an unused inode. The mode specifies permissions and can be ignored for this project. Each inode stores three extents. This allows the inode to refer to three regions of blocks that are contiguous. Should an application attempt to extend a file that already uses the three extents where the last one cannot be extended (that block is in use), your lfs implementation will need to move and coalesce extents to make room.

1.2.5 Dirent (lfs_dirent)

Dirents are data structures used to store the contents of a directory. This is stored in the data blocks of a LFS_DIRECTORY's inode.

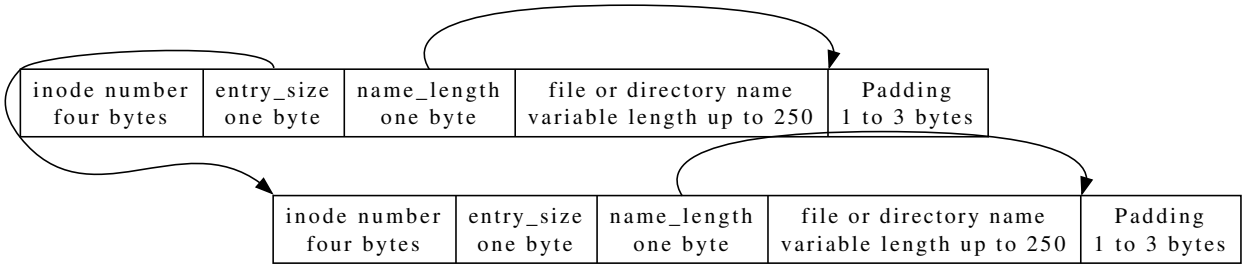


Figure caveat: the lengths are sizes, not pointers and the alignment padding isn't explicit. Figure caveat 2: the entry length of the second dirent will point beyond the end of the second dirent, even if there is nothing there because that's the end of the directory. This is explicit in the example below.

Directories are files consisting of appended records of type lfs_dirent. Note that you will be overflowing the array to write names of any length; sizeof(struct lfs_dirent) will very likely confuse. To calculate the size for a new dirent, round 4 + 2 + name_length to the nearest multiple of four. A helper function would be a good idea here.

Some code for seeking through the list in a directory is in lecture notes: to find an entry, first look for entries that match the length.

You must support deletion; We covered three obvious ways to remove from the middle of the list without rewriting the entire directory in class, so I leave it to you to determine.

When creating a directory, it is not entirely empty. Each directory has both '.' and '..' entries. The root directory will have contents:

```
01 00 00 00 04 01 2e 00
01 00 00 00 04 02 2e 2e
```

If you're wondering why inode 1 is represented using 01 00 00 00, smack yourself. (LSB.)

I encourage you to not write these 16 bytes, but instead to write a dirent constructor function to use to build the directory, since that can be reused.

1.2.6 segment

Segment is the unit that LFS keeps the buffer in the memory. One of the fundamental assumption is that memory is cheaper than before. LFS keeps a large buffer in the memory, and it does not perform actual disk IO until this buffer is filled or user requires it. Segment have size of multiple blocks, 16 in the version for GeekOS, and it contains arbitrary number of file blocks, inodes, and one segment summary at the beginning.

1.3 Buffer Cache

The buffer cache allows you to keep recently-accessed blocks in memory for a while (for example, the inode for a directory), and hold in-progress modifications to blocks until all the writes to that block are done and can be committed back. This is largely how you will interact with the filesystem in the GeekOS kernel.

Each buffer in the cache has a block number (the backing store block number), the buffer in memory, and two flags:

in use Don't touch it, it's in use. Some thread is reading from it or writing to it. Or it's going to disk. The flag is set when you Get_FS_Buffer(), and cleared when you Release_FS_Buffer(). (Releasing doesn't remove the buffer from cache, it just marks it not in use.) If the flag has been set when Get_FS_Buffer() is called, you'll block, so pretend it is a lock acquisition. If your code appears to hang while the debugger's "thread apply all bt" shows no threads running, that indicates deadlock; your code might have decided to Get_FS_Buffer() on the same block twice: e.g., to read the in-use blocks bitmap inode and the root inode, both of which are in the same block.

dirty It has been modified. Mark a buffer dirty using Modify_FS_Buffer(). The buffer cache will take care of eventually writing the block back to disk, in whatever order it chooses.

Call Create_FS_Buffer_Cache() to make a new buffer cache for the device; it is given a block size (in our case 512 bytes since we will treat disk sectors as file system blocks) at creation time.

Call Sync_FS_Buffer() to push back specific blocks, and Sync_FS_Buffer_Cache() to push them all back. Ideally, one would use Sync_FS_Buffer for flushing the blocks of a file on close or of filesystem metadata eagerly. One would use Sync_FS_Buffer_Cache to sync the whole filesystem. (This would be comparable to the "sync" program that administrators worried about a crash or shutdown invoke.)

There's a Destroy_FS_Buffer_Cache() function too. I can't see a purpose to it in this assignment.

The buffer cache must be used to prevent too many reads or writes from hitting the virtual disk device. The buffer cache must be sync'd as the shell exits before Hardware_Shutdown().

We have one sample image, `lfs.img`, included in the distribution.

I expect to add more soon, but this should be enough to start with mount and some simple development.

1.4 Part 1: Mount and Read Functions

The mount system call allows you to associate a filesystem with a place in the file name hierarchy. The Mount call is implemented as part of the VFS code we supply (see Mount function in vfs.c); you will implement your Init_LFS function so that VFS's mount code will call your function LFS_Mount() in lfs.c. Among other things, it must ensure that the filesystem being mounted is LFS.

Open files are tracked by the kernel using a struct File. Each user space process will have an associated file descriptor table that records which files the process can currently read and write. A user process can have up to USER_MAX_FILES files open at once. The file descriptor table is implemented as a struct File *file_descriptor_table[USER_MAX_FILES] array in struct User_Context. Not all the entries in the file descriptor table are necessarily open files, since usually a process has less than USER_MAX_FILES files open at once. If fileList[i] is NULL, it represents a free slot (file descriptor is not used). This descriptor will be filled out by the code in VFS; e.g., see Open in vfs.h, whose pFile argument is a pointer to a free slot in the table.

From the list of system calls below, implement all functionality for reading. Open (without the create option), Open_Directory, Close, Read, Readentry, Stat, FStat, Seek. "All functionality" includes the implementation of these functions for LFS in src/geekos/lfs.c. (The syscalls are nearly trivial, since most of the work is done for you in vfs.c, and most are implemented for you. Some implementations may not be correct for this assignment, since they are based on another file system design having a different specification.)

Ensure Makefile.linux ties the second ide disk to a valid lfs filesystem. (Add "-hdb lfs-1500k.img" to the qemu line, for example.)

1.5 Part 2: Write Functions

Implement the rest: the create option to open, delete, write, create_directory, sync. Deletion should release the blocks to be reused by another file.

1.6 New System Calls

You will implement new system calls as described below. The semantics are very similar to the UNIX file system. Some notes:

- All user-supplied pointers (e.g., strings, buffers) must be checked for validity.
- Some checks are already done by vfs.c.

Mount Takes a device name ("ide1") to be mounted at a prefix ("/d") with a given file system type ("lfs"). ENOMEM on a failure to allocate memory. EINVALIDFS if not a lfs file system. ENAMETOOLONG, ENOFILESYS handled by vfs.

Your LFS Mount function should not "validate" the filesystem settings except for magic and version fields.

Open Takes a path and a mode, returning a file descriptor. ENOMEM on a failure to allocate memory. EINVALID if user arguments are incorrect, e.g., bad flags. ENAMETOOLONG if the entire path is longer than 1024 bytes or any individual label is longer than 252 bytes. EMFILE if no more file descriptors. ENOTFOUND if the file does not exist and O_CREATE was not given, or if the containing director does not already exist. Use Allocate_File in vfs.c to get a File. The permissions values are flags that may be or'ed together in a call, e.g., (O_CREATE — O_READ — O_WRITE)

Open_Directory See Open, except return ENOTDIR if called with the name of a file.

Close EINVALID if fd out of range.

Delete Takes a path. ENOTFOUND if the path is already absent. EACCESS if path is a non-empty directory. Let "recursive" always be false (I consider recursive file system operations in the kernel to be an abomination).

Read Acts as unix read(). Returns EACCESS if file was not opened for reading. EINVALID if fd out of range, ENOMEM if no memory, ENOTFOUND if the file descriptor has not been opened or has been closed. It's fine to return fewer bytes than asked for, for example, when near the end of the file. Be sure to advance the file position.

Read_Entry Returns 0 for a valid entry. Returns VFS_NO_MORE_DIR_ENTRIES when all entries have been read. (Note: this return convention (zero on something read) is inconsistent with the convention for Read (zero on EOF).) Error codes as described for read.

Write Acts as unix write. Error codes as described for Read() above, with obvious edits.

Stat Similar errors as Open (except EMFILE). VFS_File_Stat is defined in fileio.h

FStat Similar errors as Read, where applicable. FStat fills in the stat structure given a file descriptor rather than a path name as in Stat.

Seek Similar errors as Read. Acts as unix seek, assuming SEEK_SET, that is, an absolute position is the offset. The behavior of seek() on lfs directories shall be undefined; the behavior of readdir after seek equally undefined. (Undefined means do what you want; I don't want to specify.)

Create_Directory Similar errors as Open. Should NOT create directories recursively; e.g. CreateDirectory("/d/d1/d2/d3/d4"), should fail if /d/d1/d2/d3 does not exist.

Sync Forces any outstanding operations to be written to disk. That is, flush dirty buffers in the buffer cache to disk.

Format Do not implement. (Other file systems in 412 may implement Format as a system call, which permits quick testing but is otherwise atypical.)

2 Testing and Requirements

2.1 Requirements

Here are some must do's for the project:

- Make sure your Mount() works well, so that we can test your project. If we cannot Mount() a LFS, we cannot grade your project. Do not refuse to mount our valid images.
- You might also want to mount /d to ide1 automatically in main() to speed up your testing, but the code you submit should not mount /d automatically.
- You should support arbitrary file sizes.

You do not need to consider: situations where two processes have the same file open, situations where one process opens the same file twice without closing it in between, or interactions with fork.

2.2 Testing

Finally, in src/user there are some programs that can be used to test your file management system calls: cp.c, ls.c, mkdir.c, mount.c, lfstst.c, touch.c, type.c.

There are other test files; these have not been updated for this file system.

Most of the submit server tests are based on lfstst.c. Any point values configured in lfstst are not necessarily used on submit.

2.3 Study questions

Consider the difference between this mini inode and a more realistic inode; what do others maintain, and why does lfs not include it?

Consider the difference between the original unix file system and FFS; in your implementation, how often do you jump between reading inodes and reading data blocks?

Consider readent (SYS_READENTRY); does it make sense to have this be a system call? "strace ls" on a linux machine. What system call does it use instead? Check the man page and explain why that's a better design.

Using your experience from this project, what does it mean for a filesystem to have been cleanly unmounted? How does the kernel know if a filesystem is clean when it is being mounted? What would have to be added to this project to allow that decision? How would one check and repair LFS?