Project 5: GeekOS File System 2

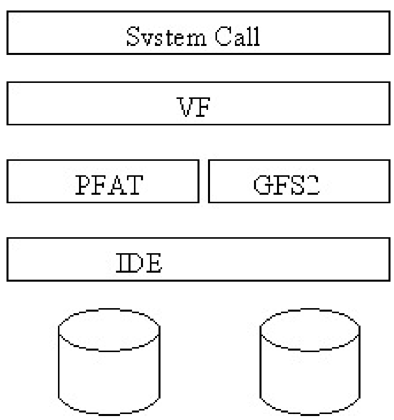
CMSC412 Spring 2013

1 Overview

The purpose of this project is to add a writable filesystem, GFS2, to GeekOS. Unlike the existing PFAT, GFS2 includes directories, inodes, direntries, etc. The new filesystem will reside on the second IDE disk drive in the emulator. The PFAT drive will continue to hold user programs while you format and test your implementation of GFS2.

1.1 VFS Introduction

Since GeekOS will have two types of filesystems (PFAT and GFS2), it will have a 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 GFS2 routines when a file operation refers to a file in GFS2.



The implementation of PFAT is in pfat.c. You will implement GFS2 in gfs2.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_gfs2FilesystemOps, s_gfs2MountPointOps, s_gfs2DirOps, and s_gfs2FileOps in gfs2.c. You should also add a call to Init_GFS2 provided in gfs2.c to main.c to register the GFS2 filesystem. In general, use the PFAT implementation as your guide to interfacing GFS2 with VFS.

1.2 GFS2 Filesystem Data

1.2.1 Superblock (gfs2_superblock)

magic: 0x47465332
version: 0x00000100
block_size 512, 1024, 4096
blocks_per_disk
block_with_inode_zero
number_of_inodes
replica_superblock_addresses

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 1024 (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.

A replica of the superblock will, when you format, be added at blocks_per_disk / 2. That's it. On format, just one. Note that this is a block offset, which is different from the byte offset of the main superblock. All the other replicas specified in the superblock will be at block "0", which means not replicated.

1.2.2 Primitive types

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

gfs2_blocknum The number of the block (not sector) on disk. Blocks are of size 512, 1024 or 4096 bytes depending on the disk image (see superblock).

gfs2_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.

1.2.3 Inode (gfs2_inode)

Inodes are data structures that store information about a file or directory on the filesystem. Their structure in GFS2 is as follows:

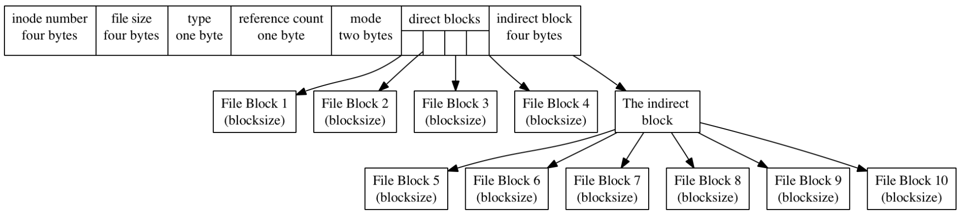


Figure note: I show ten blocks; the number of indirect blocks that can be addressed (and thus the maximum size of any file) depends on the size of the block.

The inode number is implied by location, but is specified explicitly as well. This is set to the current inode if the inode is in use, otherwise it is used to point to the next free inode in the list. The size of the inode is constrained so that an even number of inodes fit into each block (inodes won't span blocks).

The type can be either:

GFS2_DIRECTORY It's a directory. It won't be read or written directly by applications.

GFS2_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 direct blocks as well as an indirect block. The direct block array is a list of GFS_DIRECT_BLOCKS (in this case, 4) gfs2_blocknums. These block numbers point to up to the first four blocks of the file or directory, where the data is stored. If a file or directory requires more than 4 blocks (as determined by the file size), then a separate block is allocated which stores a list of more data block numbers.

1.2.4 Dirent (gfs2_dirent)

Dirents are data structures used to store the contents of a directory. This is stored in the data blocks of a GFS2_DIRECTORY's inode.

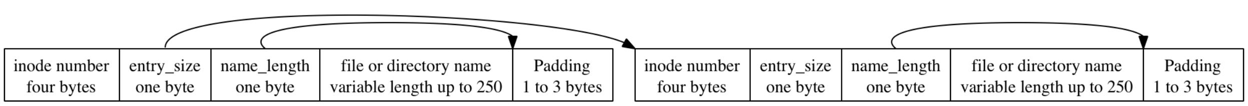


Figure caveat: the lengths are sizes, not pointers and the alignment padding isn't explicit.

Directories are files consisting of appended records of type gfs2_dirent. Note that you will be overflowing the array to write names of any length; sizeof(struct gfs2_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 do have to support deletion; I believe there is an obvious way to remove from the middle of this list without rewriting the directory, 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.

1.2.5 Free Blocks Bitmap

Track free disk blocks using a bit vector. A library called bitset is provided (see bitset.h and bitset.c) that manages a set of bits and provides functions to find bits that are 0, which correspond to free disk blocks (note that this is the opposite of the textbook, which uses a 1 to indicate a free block).

The free disk blocks map is stored within the file system using inode 2. On format, create this file, filled with ones for every block in use, i.e. those allocated to inodes, for the block consumed by the root directory, for the block containing the free blocks bitmap, and for the block containing the replica superblock(s).

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.

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 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. (i.e., you've run the "sync" program before, right?)

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

1.4 Part 1: Format

You will implement the code for format outside geekos, using src/tools/gfs2f.c. The gfs2f tool will function much like the buildFat.c tool that packages your user code and pagefile into diskc.img.

Your gfs2f must take the following options:

output file name Obvious. Create if it doesn't exist. Overwrite if it does.

block size Your format tool must support block sizes of 512, 1024, and 4096 bytes.

number of blocks Your format tool must be able to construct filesystems of at least 100 MB.

Your gfs2f may accept the following additional options:

names of files to include You may appreciate having some files to read in part 2. These files should go in the root directory.

Inode balance: The number of inodes in a disk image is fixed. We need to clearly specify this to predict how much free space is on disk and how many files our filesystem will support. The number of inodes should be computed so that there are no more than 10 blocks per inode. Take the number of blocks, divide by ten, round up to determine the minimum number of inodes. Then we divide this minimum number of inodes by the number of inodes that will fit in a block and round up to determine how many blocks will be used by that minimum number of inodes. Finally, the number of inodes in the system will be the number that fill that many blocks.

$$\text{num inodes} = \left\lceil \frac{\left\lceil \frac{\text{num blocks}}{10} \right\rceil}{\frac{\text{block size}}{\text{inode size}}} \right\rceil * \frac{\text{block size}}{\text{inode size}}$$

Key steps:

- Create a valid, but empty directory in inode 1 with self inode pointer 1. Mark its ref count as 1. This is the root directory, "/".
- Create the free blocks bitmap file in inode 2. Mark its ref count as 1.
- Initialize all the inodes. For inode 3..n, make the self inode pointer i+1, until for inode n it is zero. Ensure each of the unused inodes has reference count zero.
- Write the superblocks.

Hints:

- Use mmap() to access various parts of the file at random; you can manipulate the data layout without any writes or read/modify/writes.
- Determine the free blocks bitmap ahead of time. Inodes make blocks be not free.
- Implement dirent manipulation routines so that you can re-use them. Be sure you understand struct gfs2_dirent and why sizeof(struct gfs2_dirent) is likely useless.
- The resulting file must be readable (use 0664 as the third parameter to open()).
- Construct test files in /tmp if on linuxlab, which should be faster and is less likely to cause quota trouble.

- You need not support a file system with fewer than 10 blocks.

Sample images are at:

- `http://www.cs.umd.edu/class/spring2013/cmsc412/project5/gfs-4096x4096.img.bz2`
- `http://www.cs.umd.edu/class/spring2013/cmsc412/project5/gfs-4096x128.img.gz`
- `http://www.cs.umd.edu/class/spring2013/cmsc412/project5/gfs-512x64.img`

Larger images are compressed. These samples may not precisely match the specification but are close enough.

1.5 Part 2: 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 gInit_GFS2 function so that VFS's mount code will call your function GFS2_Mount() in gfs2.c. Among other things, it must ensure that the filesystem being mounted is GFS2.

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 *fileList[USER_MAX_FILES] array that you must add to struct User_Context. Note that not all the entries in the fileList 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 gpFile 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 GFS2 in src/geekos/gfs2.c. (The syscalls are nearly trivial, since most of the work is done for you in vfs.c.)

Modify Makefile.linux (if not done for you) so that the second ide disk is tied to an image you formatted in part 1. (Add "-hdb gfs-512x64.img" to the qemu line, for example.)

1.6 Part 3: 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.

1.7 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 ("gfs2"). ENOMEM on a failure to allocate memory. EINVALIDFS if not a gfs2 file system. ENAMETOOLONG, ENOFILESYS handled by vfs.

Your GFS2 Mount function should not "validate" the filesystem settings except for magic and version fields, and that block size is support-able (a multiple of 512, or 512/1024/4096 at least). Other items, e.g., the number and start location of inodes and the total number of blocks, should be trusted even if your formatter would not generate those values.

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 filename is longer than 1024 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.

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. If current file position is beyond the end of the file, Write should allocate only the necessary blocks for the write, leaving the file sparse.

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

FStat Similar errors as Read. 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 gfs2 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 g/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 GFS2, we cannot grade your project. Do not refuse to mount our valid image.
- 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 file sizes of between 0 and the maximum file size permitted by the block size and inode structure defined in the project.

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, gfs2tst2.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 gfs2tst2.c. Any point values configured in gfs2tst2 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 gfs2 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 GFS2?