CMSC 412 Project #5

File System

Due Thursday, May 1 at 5:00pm

Introduction

The purpose of this project is to add a new filesystem to GeekOS, as well as the standard operations for file management.

This project will be done in teams, but the teams will be different than they were for project #4. Also there are different variations of the project for each team. Make sure to get your team specific variation (it will be emailed to you).

CFS - Chameleon FileSystem

The main part of this project is to develop a new filesystem for the GeekOS. This filesystem will reside on the second IDE disk drive in the QEMU emulator. This will allow you to continue to use your existing PFAT drive to load user programs while you test your filesystem. The second IDE disk's image is called `diskd.img`.

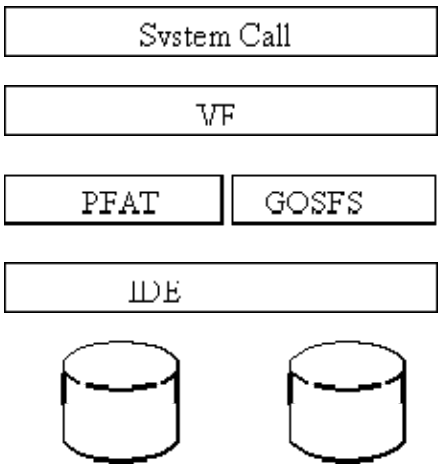
CFS will provide a filesystem that includes multiple directories and long file name support.

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.

Then you can mount the CFS file system on drive 1 onto `/d`, for instance.

VFS and file operations

Since GEEKOS will have two types of filesystems (PFAT and CFS), it has a virtual filesystem layer (VFS) to handle sending 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 CFS routines when a file operation refers a file in the CFS filesystem.



Most of the `System Call` layer is already implemented in syscall.c and the PFAT in `pfat.c`. Thus the only component you need to take care of is the CFS one.

Each user space process will have a file descriptor table that keeps track of which files that process can currently read and write. Any user process should be able to have up to 10 files open at once. The file descriptors for a user process are kept in the `files[MAX_OPEN_FILES]` array in `struct User_Context`. Note that not all the entries in the `files` are open files, since usually a process has less than 10 files open at once. If the field openFile.fsType == FS_TYPE_NONE that represents a free slot (file descriptor not used). But the good news is that file descriptor management is already implemented for you (see `Open()` function in vfs.c).

Your filesystem should support fixed length filenames (at most 64 bytes, including a null at the end for a file/directory name). A full path to a file will be no more than 1024 characters.

You should keep track of free disk blocks using a bit vector (as described in class). 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 (i.e. correspond to free disk blocks).

All disk allocations will be in units of 4KB (i.e. 8 physical disk blocks). Thus one bit in a bitset corresponds to a 4KB block. A bitset that is 8192 bits (1024 bytes) large will obviously keep track of 8192 * 4KB = 32 MB of data.

Directory Structure

See the recitation slides for details on directory structure. Each directory in CFS takes up a single disk block. The structure of the directory is defined in cfs.h. A directory is an array of `CFSfileNode` (55 elements, since they have to fit in a single 4KB block). Each filenode can represent either a file in the directory or a subdirectory.

Each file also has an inode (CFSiNode) associated with it. The inode for a directory is distinguished by the `isDirectory` bit. The location of the block that holds the data for the directory will be stored in the

first entry in the `blocks` array of the directory's filenode (hence entries `blocks[1]..blocks[7]` are unused).

Files

Unlike directories, that have a fixed size of one blocks (irrespective of how many files the hold), files can take up an arbitrary number of disk blocks. You will use a version of indexed allocation to represent the data blocks of your filesystem. The `blocks` field (CFSiNode, cfs.h) keeps track of data blocks for a file. The first eight 4KB-blocks are direct blocks, the ninth points to a single indirect block, the tenth to a double indirect block. See the recitation slides for a detailed layout.

New System Calls

You have to implement the semantics of the new system calls as described below. As you see, the semantics is very similar to the UNIX one.

- All of these functions vector through the VFS layer before you implement them at the CFS level. So the functions names are all of the form CFS_<function>. So the Mount call you implement is CFS_Mount in cfs.c

- You can look in pfat.c to see how a complete implementation of a filesystem using the VFS layer works. Be sure to look at the use of VFS functionality such as Allocate_File, which will be critical to use.

System Call User Function	**Return on success**	**Return on failure**	**Reasons for failure**	**Comment**
SYS_MOUNT Mount(char *dev, char *prefix, char *fstype)	0	-1	• a filesystem already mounted under `name` • illegal value for one of the parameters	Your 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, can be arbitrary.
SYS_OPEN Open(char *name, int permissions)	new file descriptor number	-1	• `name` does not exist (if `permissions` don't include `O_CREATE`) • `path to name` does not exist (if `permissions` include	• there's no *create* syscall, so setting O_CREATE will create the file. If the file exists, the call succeeds (return >= 0) but its data contents is not affected. • Should NOT create directories recursively if needed, e.g. `Open("/d/d1/d2/d3/xFile", O_CREATE)`, will NOT create d1 inside of d, d2 inside of

			`O_CREATE`) • `O_WRITE` and `O_CREATE` not allowed for directories, use `CreateDirectory` instead	d1, etc. if they don't exist already. If the leading path /d/d1/d2/d3 does not exist, the syscall fails, returning -1 • The permissions values are flags and may be or'ed together in a call. For example: • O_CREATE|O_READ • O_READ|O_WRITE • O_CREATE|O_READ|O_WRITE
SYS_OPEN_DIRECTORY Open_Directory(char *name)	New file descriptor number	-1	• name does not exist • name is not a directory	
SYS_CLOSE Close(int fd)	0	-1	• `fd` not within 0-9 • `fd` is not an open file	
SYS_DELETE Delete(char *name)	0	-1	• `name` does not exist • `name` is a non-empty directory	if `Delete(file)` is called and `file` is still open in other threads or even in the thread that called `Delete()`, all the subsequent operations on that file (except Close()) should fail
SYS_READ Read(int fd, char *buffer, int length)	number of bytes read	-1	• `fd` not within 0-9 • `fd` is not an open file • `fd` was not open with O_READ flag	• it's OK if `return value < length`, for instance reading close to end of file • increase the `filePos`, if successful There is special behavior when SYS_READ is called on a directory: • The data put into the buffer should be formatted as an array of dirEntry structs. • The length argument specifies the number of dirEntries to return • The return value equals the number of dirEntries read `dirEntry` is defined in fileio.h
SYS_READ_ENTRY Read_Entry(int fd, struct VFS_Dir_Entry *dirent)	1	-1	• `fd` is not a directory • file pointer is at end of	

			directory	
SYS_WRITE Write(int fd, char *buffer, int length)	number of bytes written	-1	• `fd` not within 0-9 • `fd` is not an open file • `fd` was not open with O_WRITE flag • `fd` is a directory	• increases `filePos` is successful • "Grow on write"- allocate blocks "on the fly" if past end of file
SYS_STAT Stat(char *file, fileStat *stat)	0	-1	• `file` is not found, readable	
SYS_FSTAT Stat(int fd, fileStat *stat)	0	-1	• `fd` not within 0-9 • `fd` is not an open file	
SYS_SEEK Seek(int fd, int offset)	0	-1	• `fd` not within 0-9 • `fd` is not an open file • `offset > fileSize`	`offset` is an absolute position; could be equal to `fileSize`, then `Write` appends, see above
SYS_CREATEDIR CreateDirectory(char *name)	0	-1	• `name` already exists, as file or directory • regular file encountered on the path to `name`	Should create directories recursively if needed, e.g. `CreateDirectory("/d/d1/d2/d3/d4")`, will create d1 inside of d, d2 inside of d1, etc. if they don't exist already. This operation should be atomic, in the sense that either the whole directory chain is created or no directory is created.
SYS_FORMAT Format(int drive)	0	-1	• illegal value for `drive` (it must work with 1, higher is optional) • `drive` is in use, i.e. mounted	formats a drive with CFS; don't need to support formatting with PFAT ; don't need to format in init code; so you can save your data between sessions
SYS_RENAME Rename(char *old, char *new)	0	-1	• Old file does not exist • New file does exist	Renames a file form old to new. The names will be within the same file system (i.e. we will not rename form /c/myfile to /d/myfile).
SYS_LINK Link(char *old, char *new)	0	-1	• Old file does not exist	Creates a hard link from one old to new. Hard links have separate directory entries but

			• New file does exist	share the same inode. Hard links are only within a partition/filesystem.
SYS_SYMLINK	0	-1	• Old file does not exist • New file does exist	Creates a symbolic link from old to new. The contents of the new file are the **name** of the old file. Symbolic links may span multiple partitions/filesystems.

Disk Layout

Number of Blocks (8 512 bye sectors)	**Purpose**
1	Superblock
Disk size/(4096*8)	Free blocks
(numInodes*sizeof(CFSiNode))/4096	Inodes
Rest of disk	files

A guideline is provided above. First block (0) is called SUPERBLOCK (defined in cfs.h as cfsHeader), and contains filesystem housekeeping data. Blocks >= 1 contain files and directories. It contains:

- The `Magic` number at the very beginning should be *0x20140000*. This tells you that the disk has a CFS filesystem on it. If you try to mount a drive and you don't find the magic signature, return error.
- `Size` is the size of the disk, in 4KB blocks. (32M / 4K = 8K for the example above)
- `numInodes` indicates the number of inodes that are on the disk (determined when the file system is formatted, must be at least 512 – probably much more)
- `firstInodeBlock` indicates the block number where the first inode is stored (numInodes follow in the block(s) right after that)
- `firstFreeInode` indicates the inode number of the first free inode on the disk.

When you do a `Format()`, you make a raw disk usable with CFS. That is:

1. Get drive's size, convert it in # of blocks. `IDE_getNumBlocks()` in `ide.c` tells you that.
2. Figure out `Free Blocks Bitmap` size, mark them all free.
3. Create a valid, but empty directory. That will be the root directory (inode #0)
4. Mark superblock, inodes, freemap, and block for root directory as used in the `Free Blocks Bitmap`
5. If everything went OK, write the `Magic`. Now the disk is ready to be mounted and used.

Notes

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

Too allow you to cache information, the VFS layer includes a Sync function. When the Sync function is called, all changed state needs to be saved to disk (i.e. the machine can be rebooted after it). You may choose to make all operations synchronous, in that case sync will be a no-op.

If a read() is called on a directory, the data returned should be in the form of an array of dirEntry structures. The length argument and the return value will indicate the number of entries to read and the number of entries that were read, rather than the number of bytes.

Project Variants

There are five a/b variants of the project. For each variant, your team will be assigned a specific option (a or b) that you will implement. The options are:

Case Sensitive File Names/Lower Case File Names

In the A option all file names are case sensitive.

In the B option all file names should be converted to lower case before being used.

Buffer Cache/No Buffer Cache

In the A option you will use the Buffer Cache API to access the disk drive. The relevant functions are:

```
struct FS_Buffer_Cache *Create_FS_Buffer_Cache(struct Block_Device *dev, uint_t
fsBlockSize);
int Sync_FS_Buffer_Cache(struct FS_Buffer_Cache *cache);
int Destroy_FS_Buffer_Cache(struct FS_Buffer_Cache *cache);

int Get_FS_Buffer(struct FS_Buffer_Cache *cache, ulong_t fsBlockNum, struct FS_Buffer
**pBuf);
void Modify_FS_Buffer(struct FS_Buffer_Cache *cache, struct FS_Buffer *buf);
int Sync_FS_Buffer(struct FS_Buffer_Cache *cache, struct FS_Buffer *buf);
int Release_FS_Buffer(struct FS_Buffer_Cache *cache, struct FS_Buffer *buf);
```

In the B option, use the raw IDE functions to access the disk drive. The relevant functions are:

```
int Block_Read(struct Block_Device *dev, int blockNum, void *buf);
int Block_Write(struct Block_Device *dev, int blockNum, void *buf);
int Get_Num_Blocks(struct Block_Device *dev);
```

Trash Can/Backup File

In the A option, when a file is deleted, it is moved into the directory /TRASH (which you should create as part of formatting the disk drive). If there is already a file with that name in the trash, the older one should be deleted and the newly deleted file placed in trash. The trash can does not have sub-directories so if the file `/d/dir/oldFile` is deleted, it should end up as /`TRASH/oldFile`).

In the B option, when a file (but not a directory) is opened for writing and it already exists, a backup copy of the file should be made in the same directory with the suffix .BU added to the file name. If there is already a file with the .BU suffix, it should be deleted and then the copy made of the file being opened. All file names we will use will have enough room for the .BU suffix to be added.

Symbolic Links/Hard Links

In the A option, you will add the system call SymLink to the file system. SymLink will create symbolic links to a file. The contents of a symbolic link file is the name of the file to symbolically link to. The file system knows it is a symbolic link since the isSymbolicLink field is set in the inode. When you open a file (or directory), you will check if it is a symbolic link and if so, read the contents of the file (the name of the linked file) and open that instead. This process can iterate several times until a non-symbolically linked file is reached.

In the B option, you will add the system call Link to the file system. The Link call creates a hard link to the file. In a hard link, each file has a directory entry but the point to a single shared inode. To correctly handle deleting inodes, you will need to maintain a reference count in the inode (the refCount field is provided for that purpose).

Recursive Directory Creation/Recursive File Deletion

In the A option, you will add a new mode to the Open system call (O_RECURSIVE). If this mode is set, and an attempt is made to create a file in a directory that does not exist, you will create the directory before opening the file. This call is recursive so on an empy file system and open of /d/dir1/dir2/file will first create /d/dir1 and then /d/dir1/dir2 before creating the file.

In the B option, you will implement the recursive option to the Delete system call. When true is passed to this option, a delete of a directory will recursively delete all the files in that directory before deleting the directory.

Requirements

- Make sure your `Mount()` works well, so that we can test your project. If we cannot `Mount()` a CFS, we cannot grade your project.
- You might also want to mount "/d" (dee) automatically in `Main()` to speed up your testing, but the code you submit should not mount "/d" automatically. "/c" (cee) should be mounted automatically in `Main()` though.

- You should support disk sizes of **at least** 32 MB. More than 32 MB is optional. Following the procedure described in the "How to create an arbitrary size big `diskd.img`" section above, in your submitted project, when someone types `gmake`, a 32 MB file should be created.
- You should support file sizes of **at least** 5 MB (double indirect threshold crossed, yes). More than 5 MB is optional.

Testing

As you saw at the top, in `src/user` there are some programs that can be used to test your file management syscalls: rm.c, cp,c, ls.c, mkdir.c, mount.c, nsp5test.c.