

# Chapter 10

## Project 5: A Filesystem

### 10.1 Introduction

The purpose of this project is to add a new filesystem to GeekOS.

### 10.2 GOSFS: GeekOS 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 Bochs emulator. This will allow you to continue to use your existing pfat drive to load user programs while you test your filesystem.

GOSFS will provide a filesystem that includes multiple directories, access control (via user ids), and long file name support. The access control will be added as part of the next project.

### 10.3 The Virtual Filesystem Layer (VFS)

In this project, you will work extensively with the virtual filesystem layer, commonly referred to as the VFS. This part of the kernel allows multiple filesystem implementations to coexist. A high level view of the VFS, as well as the C library and kernel subsystems it communicates with, is shown in [Figure 10.1](#).

The VFS layer works by dispatching requests for filesystem operations (such as opening, reading, or writing a file) to the appropriate filesystem implementation. The VFS works by defining several abstract datatypes representing mounted filesystem instances, open files, and open directories. Each of these datatypes contains a *virtual function table* which contains pointers to functions in the filesystem that the object belongs to. For example, `File` objects created by the GOSFS filesystem have a virtual function table whose `Read()` entry points to the function `GOSFS_Read()`.<sup>1</sup>

To give you an idea of how VFS works, here is what happens when a user process reads data from an open file in a GOSFS filesystem.

1. The process calls the `Read()` in the C library, which generates a software interrupt to notify the kernel that a system call is requested. It passes a file descriptor identifying the open file, the address of the buffer to store the data read from the file, and the number of bytes requested.
2. The kernel calls the `Sys_Read()` system call handler function. This function will look at the process's open file list (in `User_Context`) to find the `File` object representing the open file. It will also need to allocate a kernel buffer to temporarily hold the data read from the file. It will then call the kernel VFS function `Read()`.
3. The VFS `Read()` will find the address of the `Read` function in the file's virtual function table. Because the file is part of a GOSFS filesystem, this will resolve to the `GOSFS_Read()` function.

---

<sup>1</sup>If you are familiar with virtual function tables in C++ and Java, this is exactly the same idea.

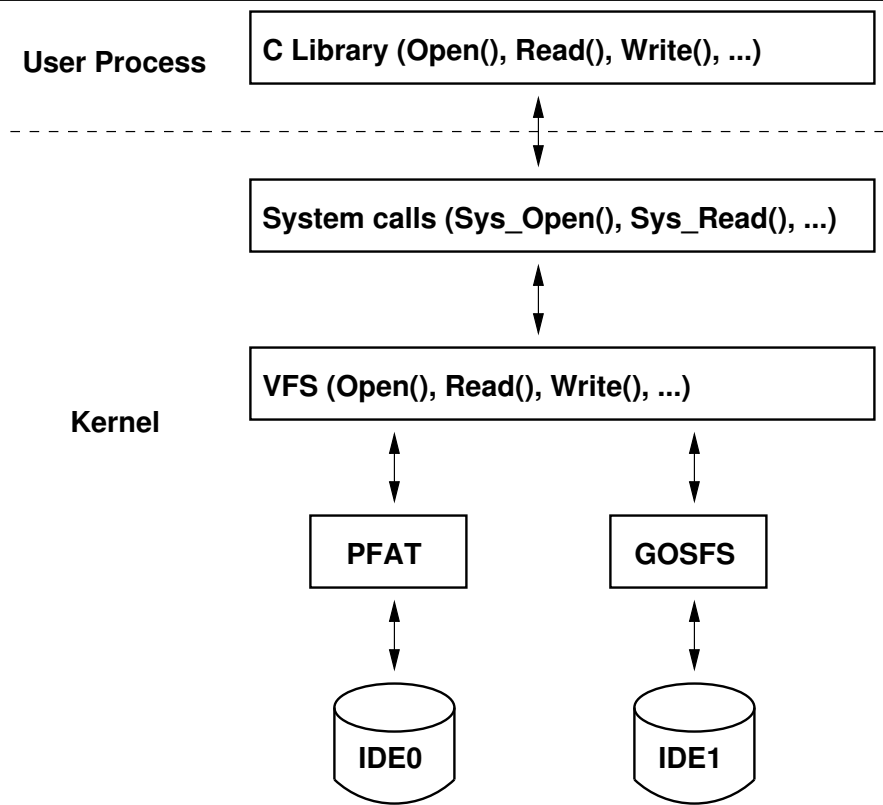


Figure 10.1: Overview of the Virtual Filesystem (VFS)

4. `GOSFS_Read()` will do whatever work is necessary to read the requested data from the file, at the current file position, copying the data into the kernel buffer created by `Sys_Read()`.
5. When control returns to `Sys_Read()`, it will copy the data read from the kernel buffer to the user buffer (using the `Copy_To_User()` function), and destroy the kernel buffer.
6. Finally, `Sys_Read()` will return, and the process will resume execution.

## 10.4 VFS Data Types and Operations

This section describes the high level VFS data types and operations, which are defined in `<geekos/vfs.h>`. You should read and understand the definitions in this header, and you may also want to look at the VFS function implementations in `src/geekos/vfs.c`.

Every operation described in this section is implemented by function in `src/geekos/gosfs.c`. For example, the `Read()` file operation is implemented by the function `GOSFS_Read()`.

### 10.4.1 Filesystem

The `Filesystem` data structure represents a filesystem type. There are two operations associated with `Filesystem`:

- The `Format()` operation formats a block device. Formatting writes filesystem metadata to the device so that it can be mounted.

- The `Mount()` operation mounts a block device containing on a particular path prefix in the overall filesystem namespace. It must check to see that the required filesystem metadata is present, and then initialize any internal data structures needed by the filesystem in order to serve requests such as opening files, reading and writing file data, etc.

### 10.4.2 Mount\_Point

The `Mount_Point` data structure represents a mounted filesystem instance. In GeekOS, filesystems are mounted on a *path prefix*, which indicates what part of the overall filesystem namespace the mounted filesystem will be part of. For example, the PFAT filesystem containing the user executables is usually mounted on the `/c` prefix. Each mounted filesystem uses a block device as its underlying storage.

`Mount_Point` objects support several operations:

- The `Open()` operation opens a file in the mounted filesystem.
- The `Create_Directory()` operation creates a directory in the mounted filesystem.
- The `Open_Directory()` operation opens a directory in the mounted filesystem, so that its entries can be read using the `Read_Entry()` operation.
- The `Stat()` retrieves the files metadata, such as size and file permissions, for a named file in the mounted filesystem.
- The `Sync()` flushes all file data and filesystem metadata stored in memory but not written to the disk. This operation is needed because filesystems usually keep data in memory buffers, and write modified data out to the disk later. Following a `Sync()` operation, all data in the filesystem is guaranteed to be written to the disk.

For all `Mount_Point` operations that take paths, the path prefix used to mount the filesystem is removed before the operation is called. So, if a GOSFS filesystem is mounted on prefix `"/d"`, and the file `"/d/stuff/foo.txt"` is passed to the high level VFS function `Open()`, the path passed to `GOSFS_Open()` will be `"/stuff/foo.txt"`.

### 10.4.3 File

The `File` data type represents a file or directory opened by a process. `File` objects are created by the `Open()` and `Open_Directory()` `Mount_Point` operations.

Regular `File` objects (i.e., those representing files, not directories) maintain a current *file position*. All reads and writes are performed relative to the current file position.

`File` objects support the following operations:

- The `FStat()` operation, like `Mount_Point's Stat()` operation, retrieves file metadata, such as size and file permissions.
- The `Read()` operation reads data at the current file position. Directories do not support this operation.
- The `Write()` operation writes data at the current file position. Directories do not support this operation.
- The `Seek()` operation changes the current file position. Directories do not support this operation.
- The `Close()` operation closes the file or directory.
- The `Read_Entry()` operation reads the next directory entry from an open directory. This operation is not supported for regular files.

## 10.5 Requirements

Each user space process will have an open file table that keeps track of which files that process can currently read and write. Any user process should be able to have 10 files open at once: this constant is defined as the `USER_MAX_FILES` macro in `<geekos/user.h>`.

The header file `<geekos/gosfs.h>` specifies constants and data structures for you to use in your filesystem implementation. All disk allocations will be in units of 4KB (i.e. 8 physical disk blocks); this value is specified by the `GOSFS_FS_BLOCK_SIZE` macro. You should maintain a free disk block list via a bit vector. A library of bitset functions is provided (prototypes in `<geekos/bitset.h>`) that will allow you to set and clear bits in blocks of memory, and also to find free bits representing free filesystem blocks. Your filesystem should support long filenames up to 127 characters, as specified by the `GOSFS_FILENAME_MAX` constant. The total file name (i.e. a full path) will be no more than 1023 characters (as specified by the `VFS_MAX_PATH_LEN` in the header `<geekos/fileio.h>`).

You will use a version of indexed allocation to represent the blocks of your filesystem. The first 8 4KB blocks are direct blocks, the next 1024 blocks are indirect blocks. These values are specified by the `GOSFS_NUM_DIRECT_BLOCKS` and `GOSFS_NUM_INDIRECT_BLOCKS` macros, respectively. The filesystem should provide a way to implement the next 1 million blocks as double indirect blocks, but you are not required to implement that interface.

## 10.6 GOSFS\_Dir\_Entry

The `GOSFS_Dir_Entry` data structure represents a single directory entry as it is stored on disk. It contains several fields:

- The `size` field stores the size of the file or directory referred to by the entry.
- The `flags` field stores several boolean flags. The `GOSFS_DIRENTRY_USED` flag indicates that the directory entry is used. If this is not set, it means the directory entry is available. The `GOSFS_DIRENTRY_ISDIRECTORY` flag indicates that the directory entry refers to a subdirectory. The `GOSFS_DIRENTRY_SETUID` flag means that the directory entry refers to a file which is a *setuid executable*: if the file is executed, the new process created will have the same user id as the file's owner. (You will not need to do anything with this bit until the next project.)
- The `filename` field stores the filename, including room for a nul terminator character.
- The `blockList` field stores the block numbers of the direct, indirect, and doubly indirect blocks representing the allocated storage for the file or directory.
- Finally, the `acl` field stores the list of Access Control List entries for the file or directory. The first entry specifies the entry for the file's owner.

## 10.7 The Buffer Cache

In your filesystem implementation, you will frequently need to read data from the filesystem into memory, and sometimes you will need to write data from memory back to the filesystem. Most operating system kernels use a *buffer cache* for this purpose. The idea is that the buffer cache contains memory buffers which correspond to particular filesystem blocks. To read a block, the kernel requests a buffer containing that block from the buffer cache. To write a block, the data in the buffer is modified, and the buffer is marked as *dirty*. At some point in the future, the data in the dirty buffer will be written back to the disk.

For this project, we have provided you will a buffer cache implementation, which you can use by including the `<geekos/bufcache.h>` header file. The `Buffer_Cache` data structure represents a buffer cache for a particular filesystem instance. You can create a new buffer cache by passing the block device to the `Create_FS_Buffer_Cache()` function. A buffer containing the data for a single filesystem block is represented by the `FS_Buffer` data type. You can request a buffer for a particular block by calling the

`Get_FS_Buffer()` function. The actual memory containing the data for the block is pointed to by the `data` field of `FS_Buffer`. If you modify the data in a buffer, you must call the `Modify_FS_Buffer()` function to let the buffer cache know that the buffer is now dirty. When your code has finished accessing or modifying a buffer, it should be released using the `Release_FS_Buffer` function.

Buffers are accessed in a *transactional* manner. Only one thread can use a buffer at a time. If one thread is using a buffer and a second requests the same buffer (by calling `Get_FS_Buffer()`), the second thread will be suspended until the first thread releases the buffer (by calling `Release_FS_Buffer()`). Note that you need to be careful never to call `Get_FS_Buffer()` twice in a row, since that will cause a self-deadlock.

The GeekOS buffer cache writes dirty buffers to disk lazily. If you want to immediately write the contents of a buffer back to the disk, you can call the `Sync_FS_Buffer()` function. It is generally a good idea to write back modified filesystem buffers when they contain *filesystem metadata* such as directories or disk allocation bitmaps; by writing these blocks eagerly, the filesystem is less likely to be seriously corrupted if the operating system crashes or if the computer is turned off unexpectedly. You can flush all of the dirty buffers in a buffer cache by calling the `Sync_FS_Buffer_Cache()`; this is useful for implementing the `Sync()` operation for your mounted filesystems.

The `Destroy_FS_Buffer_Cache()` function destroys a buffer cache, first flushing all dirty buffers to disk. This may be useful in your implementation of `GOSFS_Format()`.

## 10.8 Getting Started

Implementing a filesystem is complex. This section offers some suggestions on how to approach the project.

First, implement the `GOSFS_Format()` function. You may want to create a temporary `FS_Buffer_Cache` object to use to perform I/O.

Next, implement the `GOSFS_Mount()` function. This function is passed a `Mount_Point` object with a pointer to the block device containing the filesystem image. You will probably want to create an auxiliary data structure to store in the mount point; you can use this data structure to store a pointer to your buffer cache object, and any other information you will need when performing `Mount_Point` operations such as opening a file. You can store a pointer to the auxiliary data structure in the `fsData` field of the `Mount_Point` object.

You can test formatting and mounting a GOSFS filesystem by running the following commands from the GeekOS shell prompt:

```
$ format ide1 gosfs
$ mount ide1 /d gosfs
```

Once you can format and mount a GOSFS filesystem, you can start implementing `Mount_Point` functions. `GOSFS_Open()` is a logical place to start. When this function is called with the `O_CREATE` bit set in the mode flags, you should create a new file. As with `GOSFS_Mount()`, you will probably want to store an auxiliary data structure in the `File` object when it is created; you can use the `fsData` field for this purpose. You can use the `touch.exe` program to test file creation. You will also need to implement `GOSFS_Create_Directory()`, which you can test using the `mkdir.exe` program.

Once files and directories can be created, you can start working on `File` operations. A good place to start would be the `GOSFS_Close()` function. Eventually you will need to implement more complex functions like `GOSFS_Read()`, `GOSFS_Read_Entry()`, and `GOSFS_Write()`.

## 10.9 Issues

This section discusses some implementation issues you should think about as you work on the project.

### 10.9.1 Concurrency

Mount points, files, and directories can be accessed by multiple processes. Therefore, you will need to establish a locking discipline for your filesystem data structures to ensure that they can be safely accessed by multiple threads and processes. For this project, you can use a simple approach, such as having a single mutex protect an entire filesystem instance.

### 10.9.2 File Sharing

The `File` data structure defined in `<geekos/vfs.h>` represents a file or directory that has been opened by a process. If two processes open the same file at the same time, they will have separate `File` objects. This is necessary because both processes must have separate file positions; if one process performs a read or a seek operation, it should not affect the other process. However, both objects refer to the same file in the filesystem; changes made by one process should be seen by the other. This means that you will probably want to arrange for both VFS `File` objects to have a pointer to a common, shared data structure representing the "real" file. This arrangement is shown in [Figure 10.2](#).

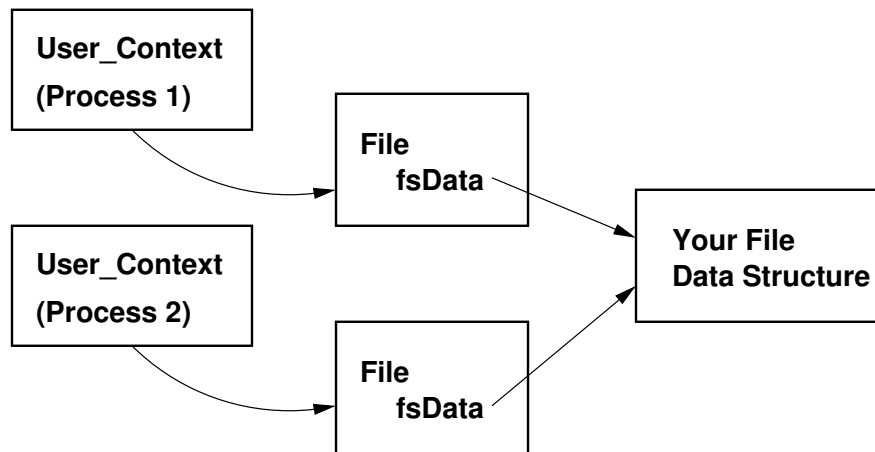


Figure 10.2: Multiple `File` objects can refer to a common data structure

Because there can be multiple references to your internal file data structure, you will probably want to keep a reference count, so you can clean up properly when there are no more references.