CS4414 (SP2017) Operating Systems

Project 4

Title: Designing Your Mini File System
Due: May 2, 2017 (12:30 p.m.)
Points: 20

The Problem

The goal of this project is to implement a mini file system on top of a virtual disk. To this end, you will implement a library that offers a set of basic file system calls (such as open, read, write, etc.) to applications. The file data and file system metadata will be stored on a virtual disk. This virtual disk is actually a single file that is stored on the "real" file system provided by the Debian operating system. That is, you are basically implementing your file system on top of the Debian file system.

You are allowed to work on the project in pairs; but groups of more than two are NOT allowed. You must generally work on the code together, with both collaborating on each segment of code (instead of working on different segments). You should ask questions for clarification if you think you have found an error or ambiguity (or even get lost) in the specification. Don't be intimidated by the length of the specification; it simply helps you write a simple (and working) version of a Linux file system that you have always used. So let's get started!

To create and access the virtual disk, you will be provided with a few definitions and helper functions in the Resources folder on Collab. **Note** that, in your library, you are NOT allowed to create any "real" files on the Debian file system itself. Instead, you have to use the provided helper functions and store all the data that you need on the virtual disk. As you can see by looking at the provided header and source files, the virtual disk has 64 blocks, and each block holds 16 bytes. You can create an empty disk, open and close a disk, and read and write entire blocks (by providing a block number in the range between 0 and 63 inclusive).

To make things easier, your file system does NOT have to support a directory hierarchy. Instead, all files are stored in a single root directory on the virtual disk. In addition, your file system does NOT have to store more than 8 files (of course, you can create and delete files and deleted files do not count against this 8 file limit). Finally, out of the 64 blocks available on the disk, only 32 must be reserved as data blocks. That is, you have ample of space to store your metadata. However, you have to free data blocks (*i.e.* make them available again) when the corresponding file is deleted. The maximum file size is 512 bytes (*i.e.* all 32 data blocks, each with 16 bytes).

Note that the maximum length of a file name is 4 letters (without a dot and an extension) and only letters are allowed to make a file name. **Note** also that file names are case sensitive. An error message must be printed to the screen if the file name entered exceeds 4 letters or characters other than letters are used to make a file name.

To manage your file system, you have to provide the following three functions.

(1) int make_fs(char *disk_name);

This function creates a fresh (and empty) file system on the virtual disk with name *disk_name*. The maximum length of the disk name is also 4 letters (and case sensitive). As part of this function, you need first to invoke *make_disk*(*disk_name*) to create a new disk. Then, open this disk and write/initialize the necessary metadata (directory, FAT, *etc*.) for your file system so that it can be later used (mounted). The function returns 0 on success and -1 when the disk *disk_name* could not be created, opened, or properly initialized.

(2) int mount_fs(char *disk_name);

This function mounts a file system that is stored on a virtual disk with name *disk_name*. With the mount operation, a file system becomes "ready for use". You need to open the disk and then load the metadata that is necessary to handle the file system operations that are discussed below. The function returns 0 on success and -1 when the disk *disk_name* could not be opened or when the disk does not contain a valid file system (that you previously created with *make_fs*).

(3) int dismount_fs(char *disk_name);

This function dismounts your file system from a virtual disk with name *disk_name*. As part of this operation, you need to write back all metadata so that the disk persistently reflects all changes that were made to the file system (such as new files that were created, data that was written, etc.). You should also close the disk. The function returns 0 on success and -1 when the disk *disk_name* could not be closed or when data could not be written to the disk.

It is important to observe that your file system must provide persistent storage. That is, assume that you have created a file system on a virtual disk and mounted it. Then, you create a few files and write some data to them. Finally, you dismount the file system. At this point, all data must be written onto the virtual disk. Another program that mounts the file system at a later point in time must see the previously created files and the data that was written. This means that whenever *dismount_fs* is called, all metadata and file data (that you could temporarily have only in memory; depending on your implementation) must be written out to the disk.

In addition to the management routines listed above, you are supposed to implement the following file system functions (which are very similar to the corresponding Linux file system operations). These file system functions require that a file system was previously mounted.

(4) int fs_create(char *name);

This function creates a new file with name *name* in the root directory of your file system. The file is initially empty. Also, there can be at most 8 files in the directory. Upon successful completion, a value of 0 is returned. The function returns -1 on failure. It is a failure when the file with *name* already exists, when the file name is too long (it exceeds 4 letters), or when there are already 8 files present in the root directory. **Note** that to access a file that is created, it has to be subsequently opened.

(5) int fs_open(char *name);

The file specified by *name* is opened for reading and writing, and the file descriptor corresponding to this file is returned to the calling function. If successful, *fs_open* returns a non-negative integer, which is a file descriptor that can be used to subsequently access this file. **Note** that the same file (file with the same name) can<u>not</u> be opened multiple times. When this happens, your file system must return -1. Your library must support a maximum of 4 file descriptors that can be open simultaneously. Upon successful completion, a file descriptor is returned. The function returns -1 on failure. It is a failure when the file with *name* cannot be found (*i.e.* it has not been created previously or is already deleted). It is also a failure when there are already 4 file descriptors active. When a file is opened, the file offset (file pointer) is set to 0 (the beginning of the file).

(6) int fs_close(int fildes);

The file descriptor *fildes* is closed. A closed file descriptor can no longer be used to access the corresponding file. Upon successful completion, a value of 0 is returned. In case the file descriptor *fildes* does not exist or is not open, the function returns -1. **Note** that when the file is closed successfully, all the meta-data (file size, FAT, *etc*.) pertaining to the file need to be updated.

(7) int fs_delete(char *name);

This function deletes the file with name *name* from the root directory of your file system and frees all data blocks and metadata that correspond to that file. The file that is being deleted must <u>not</u> be open. That is, there cannot be any open file descriptor that refers to the file *name*. If the file is currently open when *fs_delete* is called, the call fails and the file is not deleted. Upon successful completion, a value of 0 is returned. The function returns -1 on failure. It is a failure when the file with *name* does not exist. It is also a failure when the file is currently open (*i.e.* there exists an open file descriptor that is associated with this file).

(8) int fs_read(int fildes, void *buf, size_t nbyte);

This function attempts to read *nbyte* bytes of data from the file referenced by the descriptor *fildes* into the buffer pointed to by *buf*. The function assumes that the buffer *buf* is large enough to hold at least *nbyte* bytes. When the function attempts to read past the end of the file, it reads all bytes until the end of the file. Upon successful completion, the number of bytes that were actually read is returned. This number could be smaller than *nbyte* when attempting to read past the end of the file. However, when the function is trying to read while the file pointer is at the end of the file, the function returns zero. In case of failure, the function returns -1. It is a failure when the file descriptor *fildes* is not valid. The *read* function implicitly increments the file pointer by the number of bytes that were actually read.

(9) int fs_write(int fildes, void *buf, size_t nbyte);

This function attempts to write *nbyte* bytes of data to the file referenced by the descriptor *fildes* from the buffer pointed to by *buf*. The function assumes that the buffer *buf* holds at least *nbyte* bytes. When the function attempts to write past the end of the file, the file is automatically extended to hold the additional bytes. It is possible that the disk runs out of space while performing a write operation. In this case, the function attempts to write as many bytes as possible (*i.e.* to fill up the entire space that is left). The maximum file size is 512 bytes (which is, 32 blocks, each 16 bytes). Upon successful completion, the number of bytes that were actually written is returned. This number could be smaller than *nbyte* when the disk runs out of space. However, when the function is trying to write while the disk is full, the function returns zero. In case of failure, the function returns -1. It is a failure when the file descriptor *fildes* is not valid. The *write* function implicitly increments the file pointer by the number of bytes that were actually written.

(10) int fs_get_filesize(int fildes);

This function returns the current size of the file pointed to by the file descriptor *fildes*. Upon successful completion, the function returns the file size. In case *fildes* is invalid, the function returns -1.

(11) int fs_lseek(int fildes, off_t offset);

This function sets the file pointer (the offset used for read and write operations) associated with the file descriptor *fildes* to the argument *offset*. **Note** that *offset* can be negative, 0, or positive. However, it is an error to set the file pointer out of bounds (*i.e.* before the beginning of the file or after the end of the file). To append to a file, one can set the file pointer to the end of a file, for example, call *fs_get_filesize*(*fd*) first, then followed by *fs_lseek(fd, length)*, where *length* is the file size returned by *fs_get_filesize* (we assume the file pointer is currently pointing to the first byte in the file). Upon successful completion, a value of 0 is returned. The function returns -1 on failure. It is a failure when the file descriptor *fildes* is invalid or when the requested *offset* sets the file pointer out of bounds.

(12) int fs_truncate(int fildes, off_t length);

This function causes the file referenced by *fildes* to be truncated to *length* bytes in size. Note that *length* is a nonnegative number. If length is equal to 0, it is equivalent to emptying the file. If the file was previously larger than this new size, the extra data is lost and the corresponding data blocks on disk must be freed. It is not possible to extend a file using *fs_truncate* (This is the case when the file size is less than *length*). Upon successful completion, a value of 0 is returned and the file pointer is reset to the beginning of the file. The function returns -1 on failure. It is a failure when the file descriptor *fildes* is invalid or the requested *length* is larger than the file size.

Implementation

In principle, you can implement the file system in any way that you want (as long as 32 blocks of the disk remain available to store file data). However, it might be easier when you borrow ideas from existing file system designs. I recommend to model your file system after the FAT (file allocation table) design, although it is also possible (though likely more complex) to use a UNIX, inode-based design.

In general, you will likely need a number of data structures on disk, including a super block, a root directory, information about free and empty blocks on disk, file metadata (such as file size), and a mapping from files to data blocks.

The *super block* is typically the first block of the disk, and it stores information about the location of the other data structures. For example, you can store in the super block the whereabouts of the file allocation table, the directory, and the start of the data blocks.

The *directory* holds the names of the files. When using a FAT-based design, the directory also stores, for each file, its file size and the head of the list of corresponding data blocks. **Note** that **first-fit** placement strategy must be used to obtain an entry into the directory when needed. When you use inodes, the directory only stores the mapping from file names to inodes.

The *file allocation table* (*FAT*) is convenient because it can be used to keep track of empty blocks *and* the mapping between files and their data blocks. However, if you choose to use the FAT approach, **first-fit** placement strategy must be used to select a free entry when needed. When you use an inode-based design, you will need a bitmap to mark disk blocks as used and an inode array to hold file information (including the file size and pointers to data blocks).

In addition to the file-system-related data structures on disk, you also need support for file descriptors. A file descriptor is an integer in the range between 0 and 3 (inclusive) that is returned when a file is opened, and it is used for subsequent file operations (such as reading and writing). A file descriptor is associated with a file, and it also contains a file offset (file pointer). This offset indicates the point in the file where read and write operations start. It is implicitly updated (incremented) whenever you perform an *fs_read* or *fs_write* operation, and it can be explicitly moved within the file by calling *fs_lseek*. **Note** that file descriptors are NOT stored on disk. They are only meaningful while an application is running and the file system is mounted. Once the file system is dismounted, file descriptors are no longer meaningful. **Note** also that **first-fit** placement strategy must be used to obtain a file descriptor when needed.

Miscellaneous

(1) Only one codebase should be submitted per group. However, each individual in the group MUST still submit in **Collab**, so that we know you have finished. But your submission in **Collab** will simply be identical for both partners, plus tell us who your partner is. Failing to submit your project to **Collab** will result in losing all the points for the project.
(2) You must NOT change anything in the *disk.h* and *disk.c* files.
(3) You must NOT change anything in the function prototypes.
(4) Your functions must NOT print any error messages except -1 or 0.
(5) The only way to retrieve/update information from/to the virtual disk is *via* the *block_read*() and *block_write*() functions, respectively.
(6) You must use *fileManager.c* to name your source code file.
(7) You must run your program in the Debian VM environment for this project.
(8) You must use the GNU C/C++ development tools (e.g. gcc, g++).
(9) Turn your project in *via* **Collab** in a **tar** file consisting of (i) all headers you have created; (ii) source code file for project 4; and (iii) a write-up showing your design and implementation of the project (at least two pages long in **PDF** format). The **tar** file should be named as follows: **p4**, followed by your computing ID, and followed by the file extension (*i.e.* **tar**). For example, the **tar** file turned in by **John Smith** should be named **p4js7nk.tar**, where **js7nk** is the computing ID of **John Smith**.
(10) Submission of a hardcopy of the **tar** file in class is NOT necessary.
(11) Your program must be successfully compiled. Programs failing to compile will not get any points.
(12) A sample program will be provided (in the Resources folder on Collab) to drive and test your library functions. Your projects will be basically tested at this level when they are graded.