EC440: Project 5 - File System

Project Goals

- To implement a simple file system on top of a virtual disk
- To understand implementation details of file systems

Collaboration policy

- You are encouraged to discuss this project with your classmates and instructors. You **must** develop your own solution. While you **must not** share your thread-local storage code with other students, you **are encouraged** to share test case code that you developed to test your solution for this assignment.

Deadline

Project 5 is due on April 29, at 11:55 pm EDT.

Project Description

The goal of this project is to implement a simple 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, ...) to applications. The file data and file system meta-information 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 Linux operating system. That is, you are basically implementing your file system on top of the Linux file system.

To create and access the virtual disk, we have provided a few definitions and helper functions that you can find in *disk.h* and *disk.c* (See Given Code). Note that, in your library, you are not allowed to create any "real" files on the Linux 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 8,192 blocks, and each block holds 4KB. 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 8,191 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. Additionally, your file system does not have to store more than 64 files at a single time.

A constraint on your solution is that you are required to implement your filesystem using inode data structures.

Required Functions to Implement

Management Routines

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

```
int make_fs(const char *disk_name);
```

This function creates a fresh (and empty) file system on the virtual disk with name *disk_name*. As part of this function, you should first invoke *make_disk(disk_name)* to create a new disk. Then, open this disk and write/initialize the necessary meta-information for your file system so that it can be later used (mounted). The function returns 0 on success, and -1 if the disk *disk_name* could not be created, opened, or properly initialized.

```
int mount_fs(const 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 meta-information 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`**).

```
int umount_fs(const char *disk_name);
```

This function unmounts your file system from a virtual disk with name *disk_name*. As part of this operation, you need to write back all meta-information so that the disk persistently reflects all changes that were made to the file system (such as new files that are created, data that is written, ...). 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 (this should not happen).

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 unmount 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 **`umount_fs`** is called, all meta-information and file data (that you could temporarily have only in memory; depending on your implementation) must be written out to disk.

File System Functions

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 is mounted.

```
int fs_open(const 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 be opened multiple times. When this happens, your file system is supposed to provide multiple, independent file descriptors. Your library must support a maximum of 32 file descriptors that can be open simultaneously.

`fs_open` 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 32 file descriptors active. When a file is opened, the file offset (seek pointer) is set to 0 (the beginning of the file).

```
int fs_close(int fd);
```

The file descriptor *fd* 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 *fd* does not exist or is not open, the function returns -1.

```
int fs_create(const char *name);
```

This function creates a new file with name *name* in the root directory of your file system. The file is initially empty. The maximum length for a file name is 15 characters. Upon successful completion, a value of 0 is returned. *fs_create* returns -1 on failure. It is a failure when the file with *name* already exists, when the file name is too long (it exceeds 15 characters), or when the root directory is full (you must support at least 64 files, but you may support more). Note that to access a file that is created, it has to be subsequently opened.

```
int fs_delete(const char *name);
```

This function deletes the file with name *name* from the root directory of your file system and frees all data blocks and meta-information that correspond to that file. The file that is being deleted must not be open. That is, there cannot be any open file descriptor that refers to the file *name*. When the file is open at the time that *fs_delete* is called, the call fails and the file is not deleted. Upon successful completion, a value of 0 is returned. *fs_delete* 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 at least one open file descriptor that is associated with this file).

```
int fs_read(int fd, void *buf, size_t nbyte);
```

This function attempts to read *nbyte* bytes of data from the file referenced by the descriptor *fd* 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 (when 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 *fd* is not valid. The *read* function implicitly increments the file pointer by the number of bytes that were actually read.

```
int fs_write(int fd, const void *buf, size_t nbyte);
```

This function attempts to write *nbyte* bytes of data to the file referenced by the descriptor *fd* 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). A file size of at least 1 MiB must be supported. Extra credit will be awarded for supporting file sizes of up to 30 MiB and up to 40 MiB.

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 (when writing to a full disk, the function returns zero). In case of failure, the function returns -1. It is a failure when the file descriptor *fd* is not valid. The *write* function implicitly increments the file pointer by the

number of bytes that were actually written.

```
int fs_get_filesize(int fd);
```

This function returns the current size of the file referenced by the file descriptor *fd*. In case *fd* is invalid, the function returns -1.

```
int fs_listfiles(char ***files);
```

This function creates and populates an array of all filenames currently known to the file system. To terminate the array, your implementation should add a NULL pointer after the last element in the array. On success the function returns 0, in the case of an error the function returns -1.

```
int fs_lseek(int fd, off_t offset);
```

This function sets the file pointer (the offset used for read and write operations) associated with the file descriptor *fd* to the argument *offset*. It is an error to set the file pointer beyond the end of the file. To append to a file, one can set the file pointer to the end of a file, for example, by calling `fs_lseek(fd, fs_get_filesize(fd));`. Upon successful completion, a value of 0 is returned. *fs_lseek* returns -1 on failure. It is a failure when the file descriptor *fd* is invalid, when the requested *offset* is larger than the file size, or when *offset* is less than zero.

```
int fs_truncate(int fd, off_t length);
```

This function causes the file referenced by *fd* to be truncated to *length* bytes in size. If the file was previously larger than this new size, the extra data is lost and the corresponding data blocks on disk (if any) must be freed. It is not possible to extend a file using `fs_truncate`. When the file pointer is larger than the new *length*, then it is also set to *length* (the end of the file). Upon successful completion, a value of 0 is returned. *fs_lseek* returns -1 on failure. It is a failure when the file descriptor *fd* is invalid or the requested *length* is larger than the file size.

Given Code

We provide the following files:

- disk.h: This describes the disk interface. The functions declared here will be available in the autograder environment.
- disk.c: This implements the disk interface. You can use it in your unit tests. The autograder uses its own copy.
- fs.h: This describes the filesystem interface that you need to implement. It includes the functions described earlier in this document.

Access the given code at:
https://drive.google.com/file/d/1BfiS9VXIv_8PHfUzcpEg_bI9OfJUxBSX/view?usp=sharing

Some hints

You are required to use an inode-based design for your implementation of this assignment. There are many design decisions you can apply when making an inode file system. Consider that the constraints we have placed on disk size, file size, and file counts may enable you to avoid some of the more complex design options while still receiving full credit (though some of those options could help with the extra credit).

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 meta-information (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 inode table and the start offset of the data blocks.

The *directory* holds the names of the files. Since we are using inodes, the directory only stores the mapping from file names to inodes (and *not* other information like file size, which belongs in the inode).

Since we are using an inode-based design, you will need a bitmap to mark disk blocks as used and an inode table to hold file information (including the file size and pointers to data blocks). Since we are operating over thousands of blocks, the bitmap will not fit within a single integer number. Consider using an array of integers, where each element of the array contains multiple bits in the entire bitmap.

In addition to the file-system-related data structures on disk, you also need support for file descriptors. A file descriptor is a non-negative integer (maximum is 31 in this assignment) 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 (seek pointer). This offset indicates the point in the file where read and write operations start. It is implicitly updated (incremented) whenever you perform a **`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 unmounted, file descriptors are no longer meaningful (and, hence, should be all closed before a call to ***`umount_fs`***).

Submission Guidelines

- Your file system must be written in C and run in the Gradescope Linux environment. Submit your ***source code and makefile*** to Gradescope. When we run `make` in your submission, a fs.o object file must be produced, containing the compiled definitions of the required functions listed in this homework prompt. This file must not define a main function.
- You need to submit whichever files are required by your makefile to generate fs.o. This likely includes fs.c, fs.h, and disk.h (plus the makefile itself). You can include additional files, such as disk.c, but the autograder will use its own disk.c implementation, so extra files will likely just be ignored.
- Your final submission must also include a `README` file (README.md is also okay). If you relied on any **external resources** to help complete this assignment, note the resource and the challenge it helped resolve. **If not**, say so. Use this file to provide any **additional notes** you would like to share with instructors about your submission. Aside from that, it just needs a **short description** of the purpose of the project. But you are permitted to include additional details that *you* want in your README.

Oral Exams

There are no oral exams for this assignment. The deadline for this assignment goes to the last day of the semester, so there is no time for oral exams. Make sure anything you would have wished to communicate in oral exams is recorded in your README instead.