CS 170 Project #4: Simple File System *

1 Project Goals

The goals of this project are:

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

2 Implementing a simple file system

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 this header file and this source file. 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. In addition, your file system does not have to store more than 64 files (of course, you can create and delete files, and deleted files to not count against this 64 file limit). Finally, out of the 8,192 blocks available on disk, only 4,096 must be reserved as data blocks. That is, you have ample of space to store your meta-information. However, you have to free data blocks (make them available again) when the corresponding file is deleted. The maximum file size is 16 megabyte (all 4,096 data blocks, each with 4KB).

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. 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 when the disk disk_name could not be created, opened, or properly initialized.

*This project is inherited from professor Kruegel's Simple File System project in: https://www.cs.ucsb.edu/c̆hris/teaching/cs170/projects/proj5.html

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 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).

3. `int umount_fs(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.

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.

1. `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 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).

2. `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.

3. `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. The maximum length for a file name is 15 characters. Also, there can be at most 64 files in the directory. 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 there are already 64 files present in the root directory. Note that to access a file that is created, it has to be subsequently opened.

4. `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 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).

5. `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 (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 fildes is not valid. The read function implicitly increments the file pointer by the number of bytes that were actually read.

6. `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 16M (which is, 4,096 blocks, each 4K). 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 fildes is not valid. The write function implicitly increments the file pointer by the number of bytes that were actually written.

7. `int fs_get_filesize(int fildes);`

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

8. `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. 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 fildes is invalid, when the requested offset is larger than the file size, or when offset is less than zero.

9. `int fs_truncate(int fildes, off_t length);`

   This function causes the file referenced by fildes to to be truncated to a size of precisely length bytes. 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_truncate 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.

3 Implementation

In principle, you can implement the file system in any way that you want (as long as 4,096 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 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 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. 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. 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 31 (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 (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).

4 Submission Policy, Deliverables, and Deadline

Please follow the instructions below exactly!

1. You shall work in groups of *two*.
2. Your files must be in a directory named *fs*. The name of the library that we will test and link must be called *fs.o*, and the implementation must be done in C/C++. Please do not include the code from *disk.c* in your library. We will link our *disk.o* against your code and our test applications. Of course, you can (and should) include *disk.h*. Moreover, you can (and should) use *disk.c* for local testing.
3. All files that you need to build your library must be included (sources, headers, makefile) in that folder. We will just call make and expect that the object file *fs.o* is built from your sources.
4. Your project must compile on a CSIL machine. If you worked on a Windows machine or your laptop at home, then make sure it still works on CSIL or modify it appropriately!
5. Include a README with this project. Explain what you did in the README. If you had problems, tell us why and what.

6. Your solution must be submitted to Gauchospace before 03/20/2019 at 9:00 am. This project is going to be graded through live demos that will take place on 03/20/2019 9:00AM to 3:00PM. Teams will be asked to sign up for demos slots later.