In this project you will implement a very simple file system called **sfs**. The file system will be implemented as a library (**libsimplefs.a**) and will store files in a *virtual disk*. The virtual disk will be a simple *regular Linux file* of some certain size. An application that would like to create and use files will be linked with the library. We will assume that the virtual disk will be used by one process at a time. When the process using the virtual disk terminates, then another process that will use the virtual disk can be started. With this assumption, we will not worry about race conditions.

Interface

The library will implement the following functions that can be called by an application. These functions will be implemented in a file called **simplefs.c**. The prototypes of these functions will be included in a header file called **simplefs.h**. In simplefs.c, you can implement and use some additional functions that will not be called by applications directly. You can also define as many structures and variables as you wish.

- int **create_format_vdisk** (char *vdiskname, int m). This function will be used to create and format a virtual disk. The virtual disk will simply be a Linux file of certain size. The name of the Linux file is vdiskname. The parameter m is used to set the size of the file. Size will be 2^m bytes. The function will initialize/create an sfs file system on the virtual disk (this is high-level formatting of the disk). On-disk file system structures (like superblock, etc.) will be initialized on the virtual disk. If success, 0 will be returned. If error, -1 will be returned.

- int **sfs_mount** (char *vdiskname). This function will be used to mount the file system, i.e., to prepare the file system to be used. This is a simple operation. Basically, it will open the regular Linux file (acting as the virtual disk) and obtain an integer file descriptor. We will not use mmap. Other operations in the library will use this file descriptor. This descriptor will be a global variable in the library. If success, 0 will be returned; if error, -1 will be returned.

- int **sfs_umount** (). This function will be used to unmount the file system: flush the cached data to disk and close the virtual disk (Linux file) file descriptor. It is a simple to implement function. If success, 0 will be returned, if error, -1 will be returned.

- int **sfs_create** (char *filename). With this, an applicaton will create a new file with name filename. Your library implementation of this function will use an entry in the root directory to store information about the created file, like its name, size, etc. If success, 0 will be returned. If error, -1 will be returned.

- int **sfs_open** (char *filename, int mode). With this function an application will open a file. The name of the file to open is filename. The mode paramater specifies if the file will be opened in read-only mode (MODE_READ) or in append-only mode (MODE_APPEND). We can either read the file or append to it. A file can not be opened for both reading and appending at the same time. In your library you should have a open file table, and an entry in that table will be used for the opened file. The index of that entry can be returned as the return value of this function. Hence the return value will be a non-negative integer acting as a file descriptor to be used in subsequent file operations. If error, -1 will be returned.

- int **sfs_getsize** (int fd). With this an application learns the size of a file whose descriptor is fd. File must be opened first. Returns the number of data bytes in the file. A file with no data in it (no content) has size 0. If error, returns -1.

- int **sfs_close** (int fd). With this function an application will close a file whose descriptor is fd. The related open file table entry should be marked as free.

- int **sfs_read** (int fd, void *buf, int n). With this, an application can read data from a file. fd is the file descriptor. buf is pointing to a memory area for which space is allocated earlier with malloc (or it can be a static array). n is the amount of data to read. Upon failure, -1 will be returned. Otherwise, number of bytes sucessfully read will be returned.

- int **sfs_append** (int fd, void *buf, int n). With this, an application can append new data to the file. The parameter fd is the file descriptor. The parameter buf is pointing to (i.e., is the address of) a static array holding the data or a dynamically allocated memory space holding the data. The parameter n is the size of the data to write (append) into the file. If error, -1 will be returned. Otherwise, the number of bytes successfully appended will be returned.

- int **sfs_delete** (char *filename). With this, an application can delete a file. The name of the file to be deleted is filename. If succesful, 0 will be returned. In case of an error, -1 will be returned.

Assume, a process can open at most 16 files simultaneously. Hence your library should have an open file table with 16 entries.

File System Specification

The sfs file system will have just a single directory, i.e., root directory, so that it will be simple to implement. No subdirectories are supported. The block size is 4 KB. Block 0 (first block) will contain superblock information (i.e., volume information).

The next 4 blocks, i.e., blocks 1, 2, 3, 4, will contain the *bitmap*. Bitmap is used to manage free space. Each bit in the bitmap indicates if the related disk block is free or used. The next 4 blocks, i.e., blocks 5, 6, 7, 8 will contain the *root directory*. Fixed sized *directory entries* will be used. Directory entry size is 128 bytes. That means each disk block can hold 32 directory entries. In this way we can have at most 4x32 = 128 directory entries, hence the file system can store at most 128 files in the disk. Maximum filename is 110 characters long, incuding the null character at the end. A directory entry for a file will contain filename and a number to indetify the FCB (inode) for the file (that can be the index of the FCB in the FCB table). The next 4 blocks, i.e., blocks 9, 10, 11, 12 will contain the list of possible FCBs for the files, i.e., the FCB table (hence the FCB table will span 4 disk blocks). FCB size is 128 bytes. Hence a disk block can contain 32 FCBs. A field in an FCB will indicate whether that FCB is used for a file or not at the moment. Instead of this field, we could have an FCB bitmap indicated which FCB is used and which FCB is not. But for this project, we will not use FCB bitmap.

Indexed allocation will be used. One level index will used for each file. FCB for a file will not contain any data block number (no pointer in the FCB – this is just for simplicity). All data block numbers for a file will be included in the index node. Hence there will one index node per file. This will limit the size of the file. Max size for a file can be 4 KB x (4 KB / 4 Bytes) = 4 MB. Disk pointer size will be 4 bytes. That means a block number will be 4 bytes (32 bits) long.

The rest is upto you.

Experimentation and Report

Do some timing experiments and write a report about the results. Try to measure how long it takes to create, read, write a file. Try various sizes. Plot results. Try to draw conclusions.

In your report, you must also write the details of your file system design; the structure of entries, etc.

2 Submission

Include a Makefile. Then tar and gzip the directory.

3 Tips and Clarifications

- Your library will be a static library (.a). See Makefile provided about how a static library can be created.
- In the sample code, there is a program called “create_format.c”. It can be used to creat a virtual disk and initialize it with your file system. That program is simply calling the create_format_vdisk() function that is partially implemented in the library (simplefs.c). You should complete the implementation of that function. Then create_format.c program can be used to create a virtual disk and initialize (high level format).
- You should access the virtual disk in blocks (block by block). That is quite simple. Example is shown in simplefs.c.
- A disk block can belong to a single file only.
- You can not use mmap() to access the virtual disk file in this project.