Operating Systems - Final assignment

File Systems

1 Introduction

In this assignment, you are expected to design and implement a simple file system (SFS) that can be mounted by the user under a directory in the users machine. **This assignment along with the supporting files we will provide you will only be functional on Linux. Make sure you don't waste time coding it on a macOS and trying to run it on Linux later**.

2 Some specifications to simplify your life

The SFS introduces many limitations such as restricted filename lengths, no user concept, no protection among files, no support for concurrent access, etc. You could introduce additional restrictions in your design. However, such restrictions should be reasonable to not oversimplify the implementation and should be documented in your submission. Even with the said restrictions, the file system you are implementing is highly useable for embedded applications.
The following are the restrictions you will put on your code (this will also help you in coding and simplify your implmentation) :

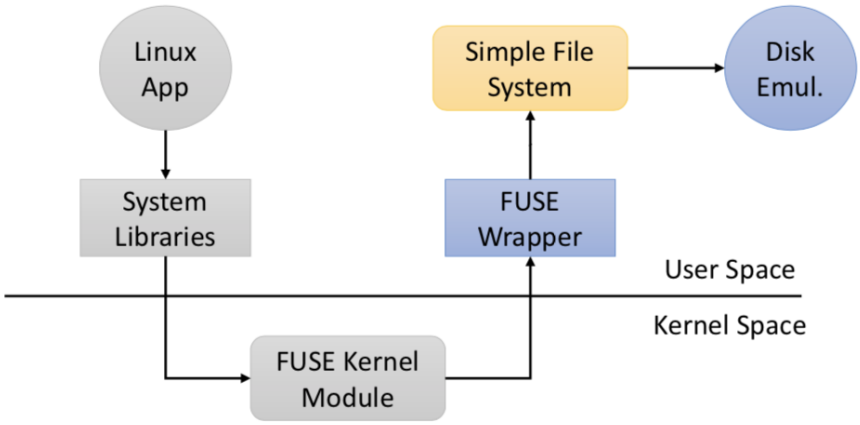
- Max file name of 16 characters.
- Max extension name of 3 characters. So in total your file name can have 20 characters (16 characters for name, 1 character for the "." period, and finally 3 characters for the extension such as "txt")
- Single level directory, so there will not be subdirectories (only a single root directory)
- You will implement an **inode** system
- You will implement **bit map** to keep track of free data blocks and find a free data block before increasing the size of a file
- The size of one data block is 1024 bytes
- The maximum number of data blocks on the disk is 1024

3 About the supporting files

We will provide you with supporting files that include the code structure of solving the assignment. You will edit the C files we give you. This is because will also provide you with test files to test your code. The test code will call functions inside your C file, so please **don't change** the main function names. You are free to implement as many helper functions and other C files as you wish (just make sure you edit your Makefile to account for your C files). In fact, through the test files, you will be able to get a sense of what your final grade for this assignment will be!

3.1 Emulated disk

Your file system is implemented over an emulated disk system, which is provided to you. The following is a schematic that illustrates the overall concept of the mountable simple file system.



The gray colored modules in the above schematic are provided by the Linux OS. The blue colored modules will be given to you. You are expected to implement the yellow colored module.

The disk emulator given to you provides a constant-cost disk (CCdisk). This CCdisk is a "virtual" disk space that is separate from your computer disk space. It can be considered as an array of sectors (blocks of fixed size). You can randomly access any given sector for reading or writing. The CCdisk is implemented as a file on the actual file system. Therefore, the data you store in the CCdisk is persistent across program invocations.

To mimic the real disk, the CCdisk is divided into sectors of fixed size. For example, we can split the space into 1024 byte sectors. The number of sectors times the size of a sector gives the total size of the disk. In addition to holding the actual file and directory data, we need to store auxiliary data (meta data) that describes the files and directories in the disk (refer to **File attributes** section in the lecture notes). The structure and number of bytes spent on meta data storage depends on the file system design, which is the concern in this assignment.

3.2 API for the assignment

As mentioned earlier, the test files we will give you will be calling your C file functions. This means your C file will be an API for the test files. We will give you the API function names and you will have to fill them. The following are the API functions you will implement:

```
void mksfs(int fresh); // creates the file system
int sfs_getnextfilename(char *fname); // get the name of the next file in directory
int sfs_getfilesize(const char* path); // get the size of the given file
int sfs_fopen(char *name); // opens the given file
void sfs_fclose(int fileID); // closes the given file
void sfs_fwrite(int fileID, char *buf, int length); // write buf characters into disk
void sfs_fread(int fileID, char *buf, int length); // read characters from disk into buffer
void sfs_fseek(int fileID, int loc); // seek to the location from beginning
int sfs_remove(char *file);// removes a file from the filesystem
```

Listing 1: Function names for your C code(you will implement this)

The following table descibes how these functions are supposed to behave

Function	Explanation
mksfs(int fresh)	Formats the virtual disk implemented by the disk emulator and creates an instance of the simple file system on top of it. The mksfs() has a fresh flag to signal that the file system should be created from scratch. If flag is false, the file system is opened from the disk (i.e., we assume that a valid file system is already there in the file system). The support for persistence is important so you can reuse existing data or create a new file system.
sfs_getnextfilename(char *fname)	Copies the name of the next file in the directory into **fname** and returns non zero if there is a new file. Once all the files have been returned, this function returns 0. So, you should be able to use this function to loop through the directory. In implementing this function, you need to ensure that the function remembers the current position in the directory at each call. Remember we have a single-level directory.
sfs_getfilesize(char *path)	Returns the size of a given file.
sfs_fopen(char *name)	Opens a file and returns the index that corresponds to the newly opened file in the file descriptor table. If the file does not exist, it creates a new file and sets its size to 0. If the file does exists, the file is opened in **append mode** (i.e., set the file pointer to the end of the file).
sfs_fclose(int fileID)	Closes a file. This means you will remove the entry from the open file descriptor table.
sfs_fwrite(int fileID, char *buf, int length)	Writes the given number of bytes of buffered data in **buf** into the open file, starting from the current file pointer. This in effect could increase the size of the file by the given number of bytes (it may not increase the file size by the number of bytes written if the write pointer is located at a location other than the end of the file).
sfs_fread(int fileID, char *buf, int length)	Reads bytes of data into **buf** starting from the current file pointer.
sfs_fseek(int fileID, int loc)	Moves the read/write pointer (a single pointer in SFS) to the given location.
sfs_remove(char *file)	Removes the file from the directory entry, releases the file allocation table entries and releases the data blocks used by the file, so that they can be used by new files in the future.

A file system is somewhat different from other OS components because it maintains data structures in memory as well as disk. The disk data structures are important to manage the space in disk and allocate and de-allocate the disk space. Also, the disk data structures indicate where a file is allocated. This information is necessary to access the file. The following is what each function will be doing

4 About inode

On-disk data structures of the file system include a super block, the root directory, free block list, and inode table. The figure below shows a schematic of the on-disk organization of SFS.

Figure 1: superblock

The super block defines the file system geometry. It is also the first block in SFS. So the super block needs to have some form of identification to inform the program what type of file system format is followed for storing the data. The figure below shows the proposed structure for the super block. We expect your file system to implement these features, but some modifications are acceptable provided they are well documented. Each field in the figure is 4 bytes long. For instance, the magic number field is 4 bytes long. With a 1024-byte long block , we can see that there will plenty of unused space in the super block.

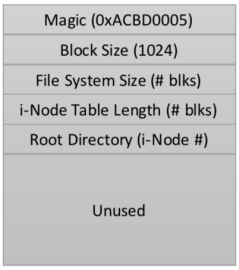


Figure 2: Super block structure

A file or directory in SFS is defined by an inode. Remember we simplified the SFS by just having a single root directory (no subdirectories). This root directory is pointed to by an inode, which is pointed to by the super block. The inode structure we use here is slightly simplified too. It does not have the double and triple indirect pointers. It has direct and single indirect pointers. With the inode all the meta information (size, mode, ownership) can be associated with the inode. So, the directory entry can be pretty simple. The figure below shows the simplified inode structure.

Figure 3: inode structure

We are suggesting the inode structure shown above to maintain a similarity to the UNIX file system. However, the simplification made to the SFS inode makes it impossible to read or write the SFS using UNIX software or vice-versa.

The directory is a mapping table to convert the file name to the inode. Remember a file name can have an extension too. You limit the extension to 3 characters max. A directory entry is a structure that contains two fields (at least): inode and file name. You could add other fields (if you find necessary). Remember the inode also has some attributes such as mode, etc. Depending on the number of entries you have in the directory, **the directory could be spanning across multiple blocks in the disk**. The inode pointing to the root directory is stored in the super block so we know how to access the root directory. We assume that the SFS root directory would not grow larger than the max file size we could accommodate in SFS.

In addition to the on-disk data structures, we need a set of in-memory data structures to implement the file system. The in-memory data structures improve the performance of the file system by caching the on disk information in memory. Two data structures should be used in this assignment: **directory table** and **inode cache**. The directory table keeps a copy of the directory block in memory. Dont make the simplification of limiting the root directory to a single block (this would severely restrict the size of the disk by limiting the number of files in disk). Instead, you could either read the whole directory into the memory or have a cache for the currently used directory block. The later one could be hard to get right.

When you want to create, delete, read, or write a file, first operation is to find the appropriate directory entry. Therefore, directory table is a highly accessed data structure and is a good candidate to keep in memory. Another data structure to cache in the memory is the free block list. In your assignment you will use **bit map**.

The figure below shows the in-memory data structure and how it connects to the other components. We need at least a table that combines the open file descriptor tables (the per-process one and system-wide one) in a UNIX-like operating system. We simplify the situation because **we assume that only one process is accessing a file at any given time**.

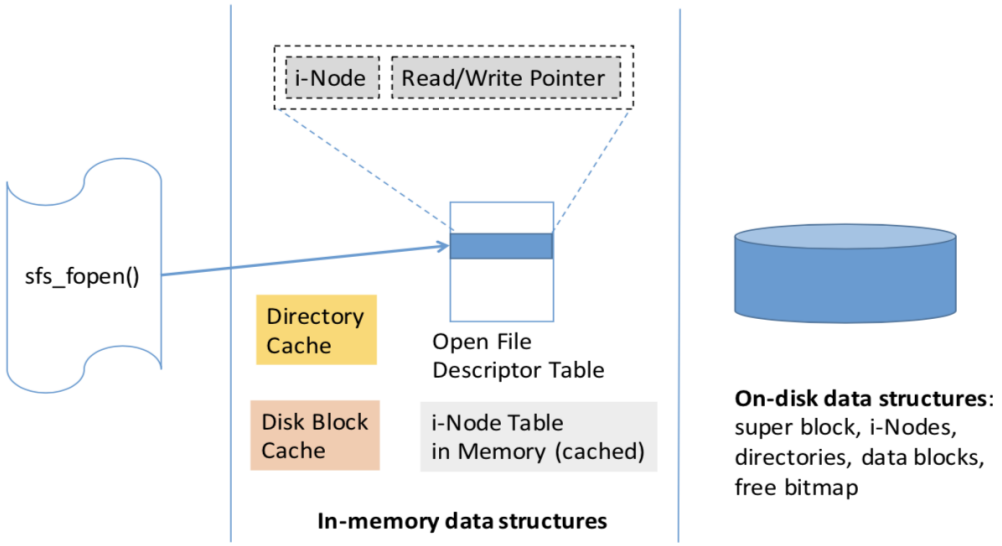


Figure 4: In-memory data structure

When a file is opened we create an entry in this table. The index of the newly created entry is the file descriptor that is returned by the file opening activity. That is the return value of the **sfs_fopen()** is precisely this index.

The entry created in the file descriptor table has at least two pieces of important information: **inode number** and a **read/write pointer**. The inode number is the one that corresponds to the file. Remember just like there is an inode for the root directory, there is one inode associated with each file. When a file is opened, that inode number is recorded in this table entry. The read/write pointer is also set according to the file system operating rule. For instance, in this assignment (SFS), you are going to set the read/write pointer to the end of the file at open so that data written into the file will be appended to the file.

In SFS, sfs_fseek() is a direct way of setting the read/write pointer value. The interesting problem you could be faced with is what to do when you perform a read or write after setting the read/write pointer. Specifically, if we have a single pointer then sfs_fread() would also advance the write pointer, and similarly, the sfs_fwrite() would advance the read pointer as well. We can simplify the complexity and let it be that way. You could opt to implement the SFS with two independent read and write pointers as well. In that case, the sfs_fseek() needs to have a parameter to specify whether the read, write, or both pointers should be moved by the seek operation.

As shown in the figure above, we have in-memory data structures and on-disk data structures in a file system. The in-memory data structures are activated as soon as the file system is up and running and they are updated every time a file system operation is carried out. While designing and implementing a given file system operation you need to think of the actions that should be carried out on the in-memory and on-disk data structures. In addition to the Open File Descriptor Table, we have variety of different caches for inodes, disk blocks and the root directory. Your design could implement all of them or some of them. File system performance is not a concern for this assignment correct operation is what we need.

5 Suggested steps for your code

5.1 Creating a file

- Allocate and initialize an inode. You need to somehow remember the state of the inode table to know which and inode could be allocated for the newly created file. Simply remembering the last inode used is not correct because as you delete files, some inodes in the middle of the table will become unused and available for reuse.
- Write the mapping between the inode and file name in the root directory. You could simply update the memory and disk copies.
- When no disk data block allocated set the file size to 0. This can also open the file for transactions (read and write). Note that the SFS API does not have a separate create() call. So you can do this activity as part of the open() call.

5.2 Writing to a file

- Allocate disk blocks (mark them as allocated in your free block list).
- Modify the file's inode to point to these blocks.
- Write the data the user gives to these blocks.
- Flush all modifications to disk.
- Note that all writes to disk are at block sizes. If you are writing few bytes into a file, this might actually end up writing a block to next. So if you are writing to an existing file, it is important you read the last block and set the write pointer to the end of file. The bytes you want to write goes to the end of the previous bytes that are already part of the file. After you have written the bytes, you flush the block to the disk.

5.3 Seek on a file

- Modify the read and write pointers in memory. There is nothing to be done on disk.

6 What to submit & test files

We have given you a Makefile, disk emulator (C and Header), SFS test files, and FUSE wrappers. The Makefile shown below has three configurations. The first two tests use hand coded test files to test your implementation. The last test (FUSE wrappers) is a full test that uses kernel functions to manipulate your file system.

```
CFLAGS = -c -g -Wall -std=gnu99 `pkg-config fuse --cflags --libs`
LDFLAGS = `pkg-config fuse --cflags --libs`

# Uncomment one of the following lines to run the corresponding scenario
#SOURCES= disk_emu.c sfs_api.c sfs_test.c sfs_api.h
#SOURCES= disk_emu.c sfs_api.c sfs_test2.c sfs_api.h
#SOURCES= disk_emu.c sfs_api.c fuse_wrappers.c sfs_api.h

OBJECTS=$(SOURCES:.c=.o) EXECUTABLE=First_Lastname_sfs

all: $(SOURCES) $(HEADERS) $(EXECUTABLE)

\$(EXECUTABLE): $(OBJECTS)
    gcc $(OBJECTS) $(LDFLAGS) -o $@

.c.o:
    gcc $(CFLAGS) $< -o $@
clean:
    rm -rf *.o *~ $(EXECUTABLE)
```

You will be editing **sfs_api.c** and **sfs_api.h**. So you will be submitting all the files we give. You are free to change the Makefile however you like. This means if you want to structure your code into more C files, that is fine, just remember to edit the Makefile to work appropriatly with your code if it needs to.

7 Grading

Functionality	**Grade**
Illustrated design (pseudo code, UMLs, your favorite strategy). You don't need to do this if your code is working. If your code does not pass any test, this is a must.	20%
File creation, read/write working, able to retrieve previously written data, but does not pass any test.	40%
Random reads and writes working with seeking, deletes working and file persistence working	50%
Passes test 1	60%
Passes test 2	75%
Passes Fuse test	90%
Memory leaks handled	95%
Excellent code structure and comments (this will show us your understanding of what functions do, and how your functions work together to achieve the required functionality)	100%