McGill

Programming Assignment #3: Mountable Simple File System

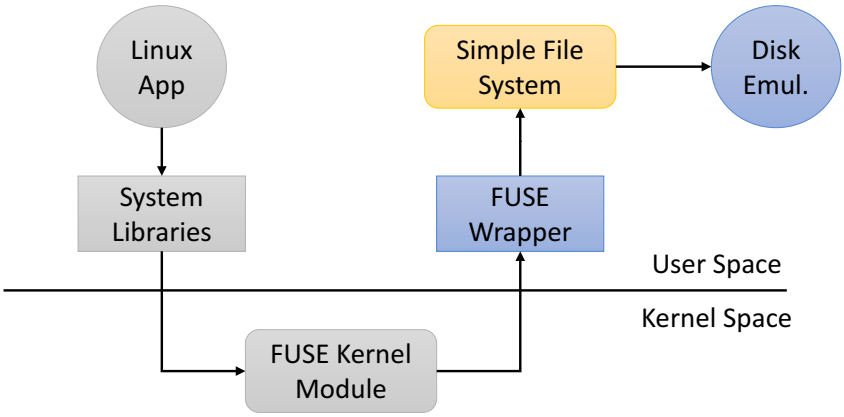
Due date: Check My Courses`

1. What is required as part of this assignment?

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 user's machine. **You need to demonstrate the SFS working in Linux.** 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. Here is a list of restrictions of the simple file system as specified in the handout:

- Limited length filenames (select an upper limit such as 16)
- No subdirectories (only a single root directory – this is a severe restriction – relaxing this would enable your file system to run many applications)
- Your file system is implemented over an emulated disk system, which is provided to you.

Here 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 are given to you as part of the support code provided as part of the assignment. You are expected to develop the yellow-colored module.

2. Objectives in detail

The SFS file system needs to have the following API. The test suite that is provided to you works with this API. If you deviate from this API, you need to change the test suite and submit the modified test suite. All the support files are provided in C and you are strongly suggested to use that language because the grader is not expected to be proficient in another language such as C++ or Rust.

```
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
```

```
int sfs_fclose(int fileID);              // closes the given file
int sfs_fwrite(int fileID,
        char *buf, int length);          // write buf characters into disk
int sfs_fread(int fileID,
        char *buf, int length);          // read characters from disk into buf
int sfs_fseek(int fileID,
        int loc);                        // seek to the location from beginning
int sfs_remove(char *file);              // removes a file from the filesystem
```

The **`mksfs()`** 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.

The **`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 in SFS we have a single-level directory. The **`sfs_getfilesize(char *path)`** returns the size of a given file.

The **`sfs_fopen()`** 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 (as part of this you create the directory entry and also allocate the i-Node for the file). If the file exists, the file is opened in append mode (i.e., set the file pointer to the end of the file). The **`sfs_fclose()`** closes a file, i.e., removes the entry from the file descriptor table (note that the file remains on disk – it is jus the association between the process and the file is terminated). On success, **`sfs_fclose()`** should return 0 and a negative value otherwise. The **`sfs_fwrite()`** writes the given number of bytes of data in **`buf`** into the open file, starting from the current file pointer. This in effect could increase the size of the file by at most 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). The **`sfs_fwrite()`** should return the number of bytes written. The **`sfs_fread()`** follows a similar behavior. The **`sfs_fseek()`** moves the read/write pointer (a single pointer in SFS) to the given location. It returns 0 on success and a negative integer value otherwise. The **`sfs_remove()`** removes the file from the directory entry, releases the i-Node and releases the data blocks used by the file (i.e., the data blocks are added to the free block list/map), so that they can be used by new files in the future.

A file system is somewhat different from other components because it maintains data structures in memory as well as disk which makes designing and implementing it a fun exercise. The disk data structures are important to manage the space in disk and allocate and de-allocate the disk space in an intelligent manner. Also, the disk data structures indicate where a file is allocated, which is necessary to access the file.

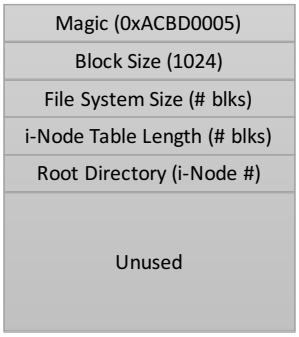
3. Implementation strategy

The disk emulator given to you provides a constant-cost disk (CCdisk). This CCdisk 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. The structure and number of bytes spent on meta data storage depends on the file system design, which is the concern in this assignment.

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

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 used 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 (recommended size), there will plenty of unused space in the super block.



A file or directory in SFS is defined by an i-Node. Remember we simplified the SFS by just having a single root directory (no subdirectories). This root directory is pointed to by an i-Node, which is pointed to by the super block. The i-Node 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 i-Node all the meta information (size, mode, ownership) can be associated with the i-Node. So, the directory entry can be simple. The figure below shows the simplified i-Node structure.

We are suggesting the i-Node structure shown above to maintain a semblance of similarity to the UNIX file system. However, the simplification made to the SFS i-Nodes already 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 i-Node. The file name can be limited as well N characters. A directory entry is a structure that contains at least two fields (at least): i-Node and file name. If it is necessary, you could add other fields in your design. Remember that i-Node also has some attributes such as mode, etc so it is not necessary to put them in the directory entry. Depending on the number of entries you have in the directory, the directory could be spanning across multiple blocks in the disk. The i-Node 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. Should not exceed 1024 bytes

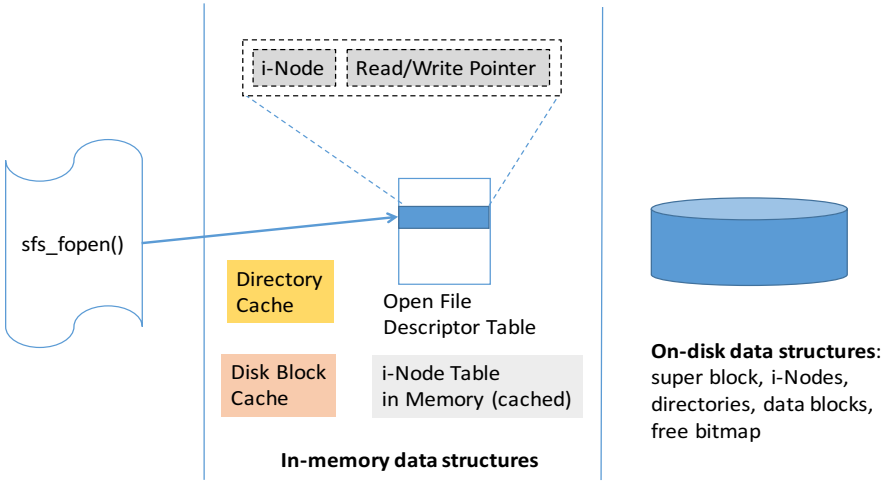
In addition to the **on-disk data structures**, we need a set of **in-memory data structures** to implement the file system. The in-memory data structures improve the performance of the file system by caching the on-disk information in memory. Two data structures can be used in this assignment: directory table and i-Node cache. The directory table keeps a copy of the directory block in memory. Don't 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 read the whole directory into the memory to make the directory interaction simple.

Further, 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. See the class notes for different implementation strategies for the free block list.

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.

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 return by `sfs_fopen()`. The entry created in the file descriptor table (same as the open file descriptor table in the figure below) has at least two pieces of important information: i-Node number and a read/write pointer. When a file is opened, we find its i-Node from the directory and record it 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 `sfs_fwrite()` would advance the "read" pointer. We can ignore this issue 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 below, 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 (or File Descriptor Table – both same because in SFS we have a single table), we have variety of different caches for i-Nodes, 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.



Rough pseudo code for creating a file:

1. Allocate and initialize an i-Node. You need to somehow remember the state of the i-Node table to know which i-Node could be allocated for the newly created file. Simply remembering the last i-Node used is not correct because as you delete files, some i-Nodes in the middle of the table will become unused and available for reuse.
2. Write the mapping between the i-Node and file name in the root directory. You could simply update the memory and disk copies.
3. No disk data block allocated. File size is set to 0.
4. 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.

Rough pseudo code for writing to a file:

1. Allocate disk blocks (mark them as allocated in your free block list).
2. Modify the file's i-Node to point to these blocks.
3. Write the data the user gives to these blocks.
4. Flush all modifications to disk.
5. Note that all writes to disk are at block sizes. If you are writing few bytes into a file, this might 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.

Rough pseudo code to seek on a file:

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

4. More About Running the File System

We have given you a Makefile, disk emulator (C and Header), SFS test files, and FUSE wrappers. The Makefile given has five configurations. The first three use hand coded test files to test your implementation. Getting your implementing running with these three test files will get you a maximum of 95% grade. If you get it working with all five tests you can get a maximum of 110% (10% bonus). Note you should edit the EXECUTABLE value to reflect your name.

Also with the assignment package, we have given a working SFS file system that uses FUSE. You can use this file system in the Trottier Lab machines. Log into the **`labX-Y.cs.mcgill.ca (e.g., lab2-2)`** machines. Create a temporary directory in the folder that contains your SFS executable (e.g., **`mytemp`**). Now run the command

```
MyFilesystem_sfs mytemp
```

Run the **`ls`** command on **`mytemp`** and you will see nothing – it is an empty directory. That is the file system is empty. Now you copy some files over there or launch an editor like vi or emacs and create some files. You will see **`fs.sfs`** file in the folder where you ran the **`MyFilesystem_sfs`** from. This file is your file system. The data in the files you copied or created are stored over here. To check the contents of the file, load it into an editor that will let you examine binary data (e.g., emacs). To un-mount the file system, find the process that is running the file system and kill it. Sometimes killing the process will leave the mount point corrupted (i.e., you cannot run the file system on the same mount point). Use the following command to clean up the corrupted mount point.

```
fusermount -u mytemp
```

5. How is the Assignment Graded?

Part I: **If you are starting this assignment few days before deadline, it is very likely that you won't complete this assignment**. The best you could do is write a very detailed pseudo code with as much detail as possible. Your pseudo code should be such that someone should be able to take that pseudo code and write a C program. **You can get up to 40% of the grade**. You will be judged on the quality and completeness of the pseudo code. There is no fixed convention on writing pseudo code – it should be highly readable (the more a pseudo code should be readable the harder it is to write!). Your pseudo code must be complete – capture all details that you consider important to create the file system (i.e., answers the questions you would have if you were coding it).

Part IIa: You have coded the SFS, but it does not run at all. Your code will be evaluated – you get up to 50%. Comment your code very well and make it super easy for the grader to go through it. If the grader is not happy, you lose marks.

Part IIb: Your program is working. It passes sfs_test0.c. You can expect around 60-70%.

Part IIc: Your program is passing sfs_test[0-1].c. You can expect around 70-80%.

Part IId: Your program is passing sfs_test[0-2].c. You can expect around 80-95%.

We have 5% for code structure, documentation, and being easy to grade. So, if you are passing all tests up to and including sfs_test2.c you are going to get 100% with a very well documented and structured code.

Part IIe: FUSE works in addition to the rest. You get the bonus 110% (100% + 10%). If you have sfs_test2.c working flawlessly, FUSE should work, unless there are some design decisions that are creating issues for FUSE. So, this is a bonus!