McGill

Programming Assignment #1: Mountable Simple File System

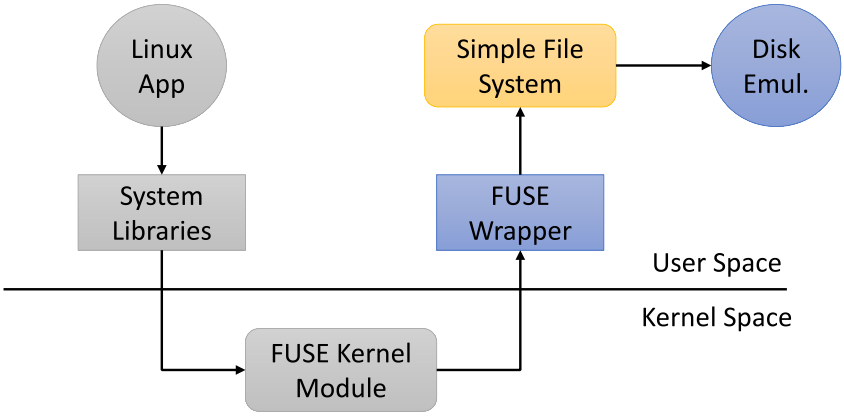
Due date: Check My Courses`

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 file system working in Linux**; however, using the same implementation you can get it working in other Unix-like operating systems (e.g., OS X) with minimal modifications. 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)
- Limited length file extensions (could be set to 3 – following the common extension length)
- 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 file system you implement will be tested in two stages. In the first stage, it is compiled against a test suite that just calls the API functions to create files, write, and read data. It performs various integrity checks for the file operations starting from simple to complex. The test suite will count errors and report you the number of errors detected while exercising the API functions. Ideally, you should debug your implementation to reach zero errors as reported by the test suite. Even in that state your implementation can have many more errors that are not detected by our test suite. We will not evaluate your implementation using a private test suite that is not released to you.

We strongly suggest that you support the following API or something that is closely equivalent to this in your implementation. You may deviate slightly, however, it is necessary to keep the functionality the same. One of the unique aspects of the file system as specified below is the independent read and write pointers. When a file is opened (we have only one mode which is read and write), the read file pointer is at the beginning of the file and write pointer is at the end of the file. That is, the file is setup for appending data to it. If the file is new, both pointers are at the beginning of the file. As you write data to the file, only the write pointer moves. As you read, the read pointer moves. So we can read a file sequentially, without manipulating (or repositioning) the Read pointer. Similarly, we can just append to a file without manipulating the Write pointer. By manipulating the Write pointer, we could write data in other places like over writing some existing data or even writing to a remote new location like shown in the code fragment below. Note that the code below does not use the API of your filesystem so it is shown for illustration purposes. If you run this in different "Unix" operating systems, you will see different behaviors!

```
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/uio.h>

int main()
{
        char buf[12];

        int fd = open("hello.dat", O_CREAT|O_RDWR);
        lseek(fd, 121323434, SEEK_SET);
        write(fd, buf, 1);
        close(fd);
}
```

You can compile and run the program and then do "ls –ls". This should show two numbers: length of the file and the number of blocks used (first number displayed). In some systems (Linux), that number would be very small (around 130). This indicates a sparse file. In OS X, it seems like a dense file where the file system is using blocks to represent many uninitialized blocks. You need to follow the sparse approach. Don't use blocks for the blank data that goes into all intermediate blocks.

The C based API for the SFS is given below. It is strongly suggested that you retain the functionality provided by the API if you decide to change it.

```
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_frseek(int fileID,
        int loc);                         // seek (Read) to the location from beginning
int sfs_fwseek(int fileID,
        int loc);                         // seek (Write) to the location from beginning
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_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 an integer that corresponds to the index of the entry for 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 exists, the file is opened in append mode (i.e., set the write file pointer to the end of the file and read at the beginning of the file). The **sfs_fclose()** closes a file, i.e., removes the entry from the open file descriptor table. On success, **sfs_fclose()** should return 0 and a negative value otherwise. The **sfs_fwrite()** writes the given number of bytes of buffered data in **buf** into the open file, starting from the current write 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). The **sfs_fwrite()** should return the number of bytes written. The **sfs_fread()** follows a similar behavior. The **sfs_rfseek()** moves the read pointer and **sfs_wfseek()** moves the write pointer 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 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.

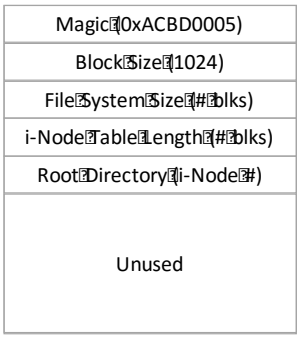
A file system is somewhat different from other components because it maintains data structures in memory as well as disk! The disk data structures are important to manage the space in disk and allocate and de-allocate the disk space in an intelligent manner. Also, the disk data structures indicate where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

3. Implementation strategy

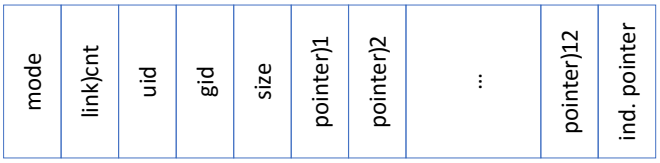
The disk emulator given to you provides a constant-cost disk (CCdisk). This CCdisk can be considered as an array of sectors (blocks of fixed size). You can randomly access any given sector for reading or writing. The CCdisk is implemented as a file on top of an underlying file system. Therefore, the 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.

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 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 (recommended size), we can see that 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 (that is the super block remembers the i-Node of the root directory). 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 pretty 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. Remember a file name can have an extension too. You can limit the extension to 3 characters max. The file name (without extension) could be limited as well (16 characters is suggested). A directory entry is a structure that contains two fields (at least): i-Node and file name. You could add other fields (if you find necessary). Remember the i-Node 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 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.

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 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

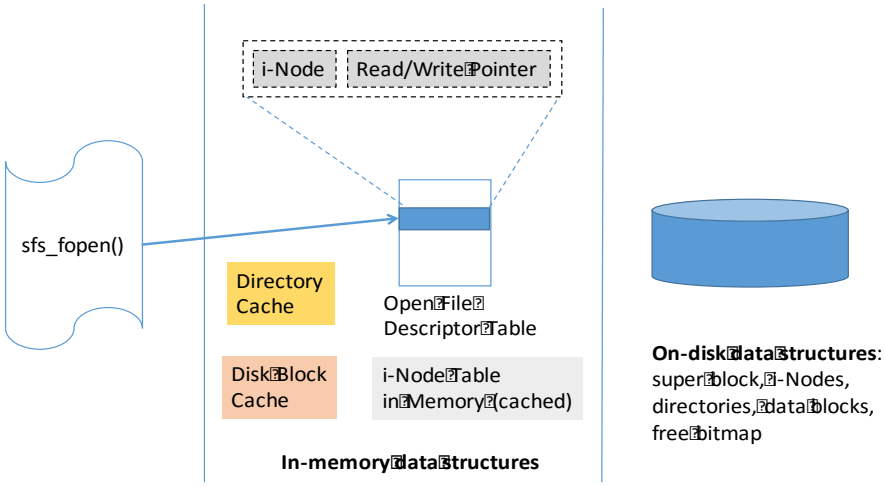
– by limiting the number of files in disk). Instead, you could either read the whole directory into the memory or have a cache for the currently used directory block. The later one could be hard to get right.

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 different in-memory data structures and how they connect 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 the file opening activity. That is the return value of the **`sfs_fopen()`** is this index. The entry created in the file descriptor table has at least three pieces of important information: i-Node number, read pointer, and write pointer. The i-Node number is the one that corresponds to the file. Remember just like there is an i-Node for the root directory, there is one i-Node associated with each file. When a file is opened that i-Node is number is recorded in this table entry. The read and write pointers are also set as specified earlier.

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 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 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.

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. What to Hand In

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 use hand coded test files to test your implementation. Getting your implementing running with these two test files will get you a maximum of 85% grade. If you get it working with all three tests you can get a maximum of 100%. We could test your implementation with other (private) tests. Note you should edit the EXECUTABLE value to reflect your name. NOTE: The last test is linking with the FUSE wrapper. So your assignment will receive full credit if it works with the FUSE wrapper.

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`** machines. Change to the **`/tmp`** directory. Create a temporary directory there (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.

```
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)
```