McGill

Programming Assignment #1: Mountable Simple File System

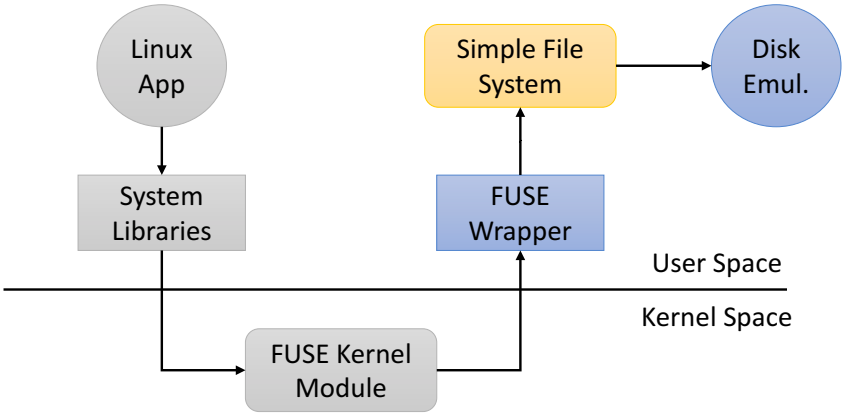
Due date: Check My Courses`

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 this working only in Linux for this assignment**; however, using the same implementation you can get it working in other operating systems 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:

- Limited length filenames (select a 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 give 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

In reality, you could implement the SFS with your own API that implements the necessary functions to interface with the FUSE wrapper provided as part of this assignment. However, for debugging purposes, we suggest that you implement your file system such that it exposes the following API. The additional test suite we provide with the assignment could be used to test your file system if you stick to the proposed API. You can deviate even significantly from the proposed API; however, in that case you will be responsible for modifying the test suite.

The suggested API for SFS is given below. The API is based on C language. 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
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 buf
void 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 the given file.

The **`sfs_fopen()`** opens a file and returns the index on the file descriptor table. If file does not exist, create the new file and set size to 0. If file exists, open the file 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 open file descriptor table. The **`sfs_fwrite()`** writes length bytes of buffered data in buf onto the open file, starting from the current file pointer. This in effect increases the size of the file by "length" bytes. The **`sfs_fseek()`** moves the read and write pointer to the given location. We have a single pointer for both purposes in SFS. The **`sfs_remove()`** removes the file from the directory entry as well as release the file allocation table entries and data blocks used by the file, so that they can be used by new files in 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.

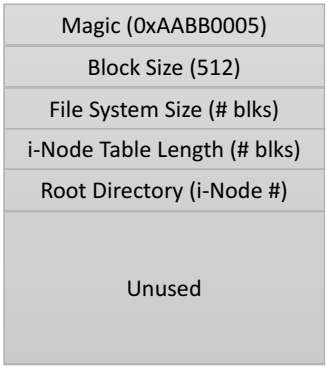
3. Implementation strategy

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 the actual file system. Therefore, the data you store in the CCdisk is persistent across program invocations. Let your CCdisk have N disk sectors with

each sector having a size of M bytes. The disk space should be used to allocate disk data structures of the file system as well the files. It is recommended that you use a sector or block size of 512 bytes.

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 512-byte long block (recommended size), we can see that there will plenty of unused space in the super block.



A file or directory is 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. 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). It is best to create a structure with i-Node, file name, file extension and store that structure in a disk block. Depending on the number of entries you have in the

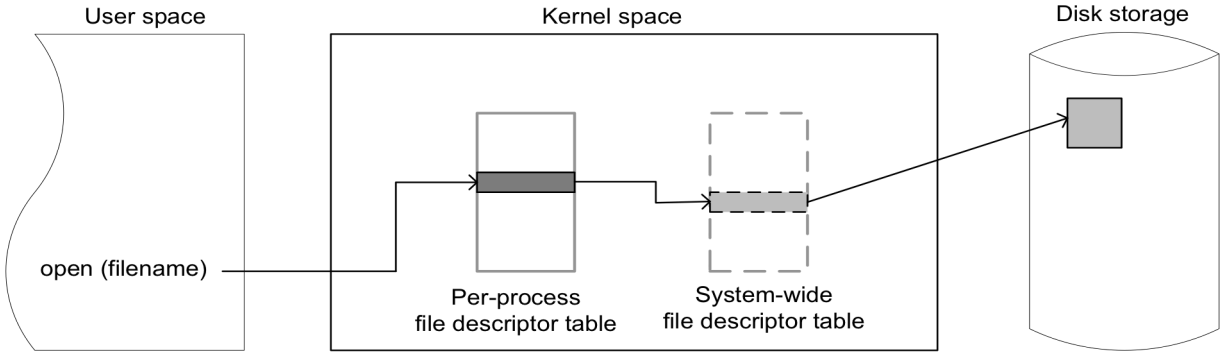
directory, the directory (only one) can have multiple blocks allocated to it. All those blocks should be pointed by the i-Node that is allocated for the root directory. We assume that the SFS root directory would not grow larger than the max file size we could accommodate in SFS.

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. 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.

Figure 2 shows an example set of in-memory data structures. The open file descriptor table(s) can be implemented in two different ways. You can have a process specific one and a system-wide one. This is more general and closely follows the UNIX implementation. You can simplify the situation and have only the table – this is reasonable because we assume that only process is accessing a file at any given time (i.e., no simultaneous access to a single file by multiple processes).

As shown in Figure 2, the entry in the file descriptor table can be used to provide some information regarding the reading and writing locations. The mandatory information is the i-Node for the file. For example in the previous example the i-Node is 3 for Test.exe. When a file is written to, the write pointer moves by the amount of bytes that is written onto the file. Normally, the write pointer would always point towards the end of the file unless it is explicitly manipulated to point elsewhere (for example, using a seek() routine in C/UNIX). The sfs_fseek() function modifies the read and write pointers. In SFS, both pointers are set by a single invocation of the sfs_fseek() function. Subsequent sfs_fread() calls and sfs_fwrite() calls change the read and write pointers independently. You should think of how to implement the read and write pointers. Please note you are required to write data in arbitrary length chunks onto the file.



Root FAT	Write ptr.	Read ptr.

Possible structure for a file descriptor table entry

Following are some of the main operations supported by the file system: creating a file, growing a file, shrinking a file, removing a file, and directory modifications.

To create file:

1. Allocate and initialize an i-Node.
2. Write the mapping between the i-Node and file name in the root directory.
3. Write this information to disk.
4. No disk data block allocated. File size is set to 0.
5. 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.

To grow 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.

To shrink a file:

1. Remove pointers to the disk blocks from the i-Node of the file.
2. Mark the disk blocks as free.

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

Please get started. We will post more information on testing and FUSE integration.

Also, these topics will be covered in the tutorials and you are strongly encouraged to take advantage of such discussions.